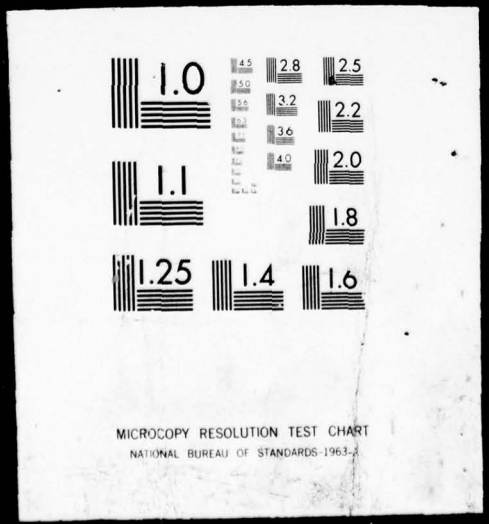
AD-A060 494 CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
THE SWITCHING STRUCTURE AND ADDRESSING ARCHITECTURE OF AN EXTEN--ETC(U)
AUG 78 R J SWAN F44620-73-C-0074
UNCLASSIFIED CMU-CS-78-138 NL

1 OF 3
AD
AD 60494

CMU-CS-78-138

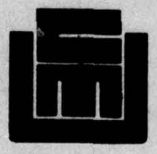
(12)
NW

The Switching Structure and Addressing Architecture of an Extensible Multiprocessor: Cm*

Richard James Swan

9 August 1978

DEPARTMENT
of
COMPUTER SCIENCE



Carnegie-Mellon University

(14) CMU-CS-78-138

(9) Interim rept.

(6) The Switching Structure and Addressing Architecture of an Extensible Multiprocessor: Cm*

(10) Richard James Swan

(11) 9 August 1978

(12) 237 p.

Carnegie-Mellon University
Computer Science Department

Submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

(13) This work was supported by the Advanced Research Projects Agency of the Department of Defense under contract F44620-73-C-0074, which is monitored by the Air Force Office of Scientific Research, in part by the National Science Foundation Grant GJ 32758X, and in part by the Office of Naval Research under contract N00014-77-C-0500. The LSI-11's and related equipment were supplied by Digital Equipment Corporation.

Abstract

Cm* is an extensible, multiprocessor computer system with a hierarchical structure. A 10 processor pilot system, constructed in the Computer Science Department, has been in operation for a year. A 50 processor system will be operational late in 1978. The hardware structure will support on the order of 10,000 processors. A major overall design objective is to efficiently support close cooperation between large numbers of concurrently executing processes.

An important component of multiprocessor systems is the switching structure which allows processors to access shared memory. The effective, maximum size (in number of processors) of multiprocessors described in the literature is limited. This limitation comes either through saturation of the access path to shared memory, or through the rapid growth of switch cost. The hierarchical switching structure of Cm* offers, in principle, indefinite extensibility of processing power, memory capacity and communication bandwidth. The cost of the interconnection structure grows approximately linearly with system size. Effective use of the structure depends on suitable decompositions of applications. We give the motivations for the switching structure and describe numerous techniques used to allow a high performance, cost-effective, deadlock free, switch realization.

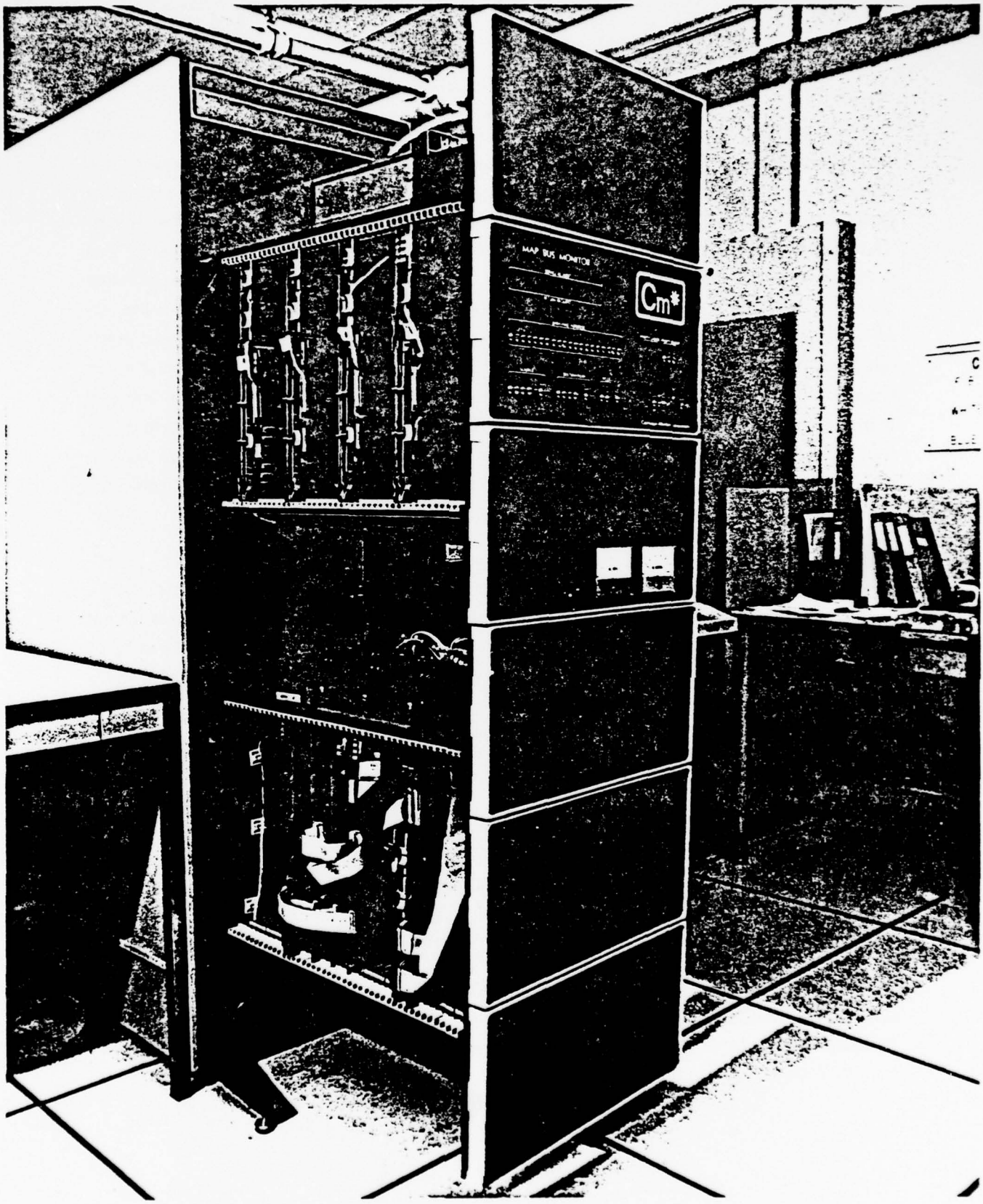
The addressing architecture of a computer system has a strong influence on its programmability. The addressing architecture is important both at the level of naming operands, for example distinguishing between stack and register oriented designs, and in naming larger conceptual entities, such as segments. The former level is determined primarily by the instruction set design. The latter level is primarily determined by the operating system. The work in this thesis is concerned with understanding addressing architecture at the level of naming and sharing objects. An object is the smallest logical collection of information (for example a segment) which is independently manipulated by the operating system and can be directly accessed by a program.

We present a simple model which allows the systematic analysis and comparison of addressing architectures. The creation of an independent addressing environment when each subprogram is invoked (as in HYDRA and FAMOS) is compared to process-based addressing (as used in Multics and CAP). We show that independent addressing environments for each subprogram allows greater flexibility in the time of name binding, has fewer levels of indirection, and is better suited to multiprocessor systems.

Close cooperation and communication between processes requires the sharing of

objects--shared access to data, semaphores, mailboxes, etc. Proper support of this sharing (particularly in a system with many, concurrently executing processes) is shown to have a substantial impact on the switching structure and addressing hardware of a multiprocessor.

As viewed from an individual processor, Cm* has a non-homogeneous structure. The access path and access time from a processor to a particular memory depends on their relative position in the structure. This inhomogenity could reduce the programmability of the system. However, the Cm* addressing architecture presents to the programmer a uniform, system wide, segmented address space. Segments are typed, protected objects. All references, by both user and kernel programs, are via capabilities. Capabilities may be passed between processes via mailbox segments. This allows the protected passing of messages, including large data structures represented by a capability list, without copying. Numerous facilities are provided in the architecture to support operating system primitives. Special provision is made to allow the movement of a segment without requiring every processor to update its mapping tables.



One Cluster of Cm*

Acknowledgements

At CMU, no research project is undertaken in isolation from others. A full acknowledgement of all the contributors to Cm* would entail listing most members of the department. To all those concerned with Cm*, thank you. I have enjoyed working with all of you. Some specific technical contributions are acknowledged in the body of the thesis. However, I wish to specially thank Paul Rubinfeld and Pradeep Reddy for their cheerful and tireless work in maintaining and expanding Cm*. John Ousterhout has made a major contribution to the success of Cm*. However, I must claim all credit for partially weaning him off orange juice and onto beer. Anita Jones, and her co-workers (listed below), showed great faith and courage by setting out to build an experimental operating system for an experimental computer system with an experimental addressing architecture. Peter Hibbard, Andy Hisgen and Tom Rodeheffer have done some remarkable hackery in trying to make multiprocessors usable by mere mortals. Levy Raskin is responsible for many of the benchmark programs run on Cm* and provided the simulation results used in this thesis. My thanks to Sam Fuller and Dan Siewiorek for making the Cm* project possible.

I must also thank Sam, my advisor, for his guidance, encouragement and friendship. I look forward to continuing my fruitful working relationship with Dan Siewiorek. A special thanks to Victor Lessor and Bill Wulf for their encouragement at the right times. I must thank all my committee, Sam Fuller, Dick Eckhouse, Dan Siewiorek and Bill Wulf, for their careful reading of *a too-early draft of this thesis.* The text was prepared using Scribe. Three cheers and a rutabaga for Brian Reid.

I dedicate this thesis to Elaine and my convivial companions at the Oakland Original.

Cm* People

The following people were involved with Cm* in some way over the past three years:

Andy Bechtolsheim, Hal Bellis, Ivor Durham, Bob Chansler, Peter Feiler, Sam Fuller, Jim Gosling, Satish Gupta, Peter Hibbard, Andy Hisgen, Anita Jones, Larry Lai, John Ousterhout, Pradeep Reddy, Levy Raskin, Tom Rodeheffer, Paul Rubinfeld, Don Scelza, Karsten Schwans, Dan Siewiorek, Pradeep Sindhu, Mickey Tsao, Steve Vegdahl.

Table of Contents

1. Multiprocessors

1.1 Introduction

Extrapolation from the small to the large is not always successful. Principles, mechanisms and intuitions valid for small systems need not be valid for large versions of the same systems. Building methods for a row boat do not extend directly to constructing an ocean liner. At one level, the two boats have similar purposes (transport over water), similar basic components (keel, prow, stern etc.) and share important design principles (stability and buoyancy etc.). Yet building a row boat is hardly preparation for construction of the QE II.

In computer science, there are many situations where direct extrapolation from small systems to large systems fails. Small programs can be written quickly and easily, even by inexperienced programmers. Large software systems require expert design and substantially more programming effort per final line of code. Uniprocessor designs can be extended to build small multiprocessors with only minor changes to switching structure, architecture and operating system. However, designs suitable for two or four processors cannot be directly extrapolated to systems with 16 processors. Hence, the development of C.mmp (a multiprocessor with 16 processors) and its operating system, Hydra, involved many new techniques. Much of this thesis is concerned with issues, in the structure and architecture of multiprocessor systems, which cannot be directly extrapolated from present systems to multiprocessors with 50 or 50,000 processors.

This chapter begins with a brief discussion of general objectives in computer design. Some factors affecting software and hardware cost are noted. Two widely applied principles for increasing the performance of computer systems are identified. One of these is the use of parallelism to exploit potential concurrency. This leads to one of the major topics of the thesis: the design of a computer system which will allow close cooperation between, on the order of, 10,000 processors. The limitations of conventional switching structures for multiprocessors are reviewed. Cm*[1] is an operational multiprocessor system whose design forms the basis of this thesis. A hierarchical switching structure, similar to Cm*, is derived in this chapter. This provides the background for a more detailed examination of the Cm* switching structure in Chapters 2, 3 and 4.

1.2 Research in Computer Design

This thesis concerns the design of computer systems. Computer design, as with the design

[1] Pronounced see-em-star. The name is derived from PMS notation [Bell and Newell, 71] for modular computer (Cm) and the Kleene *, meaning replicated an arbitrary number of times.

of other complex systems, is not an exact science. There are no specific formulae, there are no recognized procedures for designing new computer systems. Techniques for analyzing and comparing designs are imprecise and often ambiguous. Research in computer systems, can contribute to our understanding of computer design in various ways, including:

- The identification of widely applicable design principles. Often, this is the codification of a technique previously used implicitly. It may be the recognition of the common basis of independent, diverse techniques.

- The development of new analytic methods for the evaluation and comparison of designs.

- The development of specific new techniques. A description of a novel technique is useful both because it can be reapplied in similar circumstances and because it may give insight into solving apparently unrelated problems.

- The gathering and analysis of cost and performance data from operational systems. This can both validate the design in question and provide a basis of comparison for future designs.

- Measurements of operational systems can give insight into desirable optimizations in future designs. That is, measurements from existing systems can lead to the refining of design goals for future systems.

1.3 Design Objectives in Computer Systems

The design of a computer system requires reaching a judicious balance between many interacting objectives. It is normally very difficult to distinguish between objectives which are purely functional or technical and objectives which represent economic constraints. In basic terms, every design represents a specific tradeoff between performance and cost.

Performance means not only instruction executions per second and I/O bandwidth but also some measure of reliability, ease of programming, satisfaction of physical space and energy requirements, etc. Cost is perhaps even more difficult to define and measure than performance. For a specific, physically realized configuration it may be reduced to the dollar cost of the components and labor required to construct the system. However, in a research or product development situation this kind of cost measure is not very useful. To provide perspective on a design it is necessary to compare it with other systems built in different countries, at different times with widely varying technologies. Also, it is important, to be able to compare its cost with other hypothetical machines for which implementation costs are not available.

Computer manufacturers, and often, by extension, computer design researchers, are interested in the fully amortized cost of a full range of system configurations. The evaluation

of a manufacturer's real cost for a computer system is clearly very complex and well beyond the scope of this thesis. However, it is possible to informally discuss some of the factors that directly effect system cost. The other half of the equation, system performance, will be discussed in subsequent sections.

1.3.1 Software Costs and Operating System Support

Software development is an important cost factor in any computer system; in many circumstances it is the dominant factor. The economic significance of software costs versus hardware costs clearly depends strongly on the total number of systems which will use the software. To a first approximation, the marginal cost of an additional identical system is purely in the hardware because the replication cost of software is near zero. For some system types, particularly general purpose computers, the software maintenance costs may overwhelm both the initial software and hardware costs.

In conventional, general purpose machines it is only the operating system which must deal with the bare hardware. The objective of an operating system may be described as providing a comfortable high level virtual machine for the execution of user programs.[1] Design decisions at the architectural and structural level will effect the programmer--and hence software costs--at every level. However, we can argue that it is particularly important for a design to support the operating system and other system software:

1. Architectural and structural design has the greatest effect on the system software because it must execute on the bare hardware. Applications programmers are shielded by operating system functions, compilers, debuggers, etc.

2. The above can be restated: There is a certain level of virtual machine which should be presented to user programs. The closer the underlying hardware is to this virtual machine the simpler, smaller and faster can be the operating system.

3. On many systems the operating system is by far the largest single class of user. Thus it is important to optimize its performance.

4. Where frequently used primitives may be either provided within an application or by the system (i.e., shared by multiple applications) it is usually better to provide them as part of the system. (E.g., interprocess communication protocols, SIN routine, etc.) Shared access to these primitives means not only that they need only be written once and consume one set of resources but also that the standardization of interfaces will lead to better programs. For sharing of primitives to be effectively utilized they must be efficient and cheap to invoke--otherwise highly specific and optimized versions will be provided within each

[1] For example, user programs typically invoke high level operations for I/O, memory allocation, interprocess communication, scheduling, etc.

application.

5. Computer manufacturers are particularly concerned with system software as this represents the bulk of their direct software costs for a new system. It also provides the basis for all other software developed for a system.

Particular design issues, relating to the support of the operating system, will recur throughout the remainder of this thesis. The provision and manipulation of addressing environments is a crucial aspect of most operating systems. This is discussed extensively in Chapters 5 and 6.

1.3.2 Hardware Costs and the Importance of Modularity

Traditionally, digital circuit design has been concerned with the minimization of circuit elements to optimize cost. However, in practice the cost factors are usually recognized to be more complex than a simple count of the active circuit elements.

Although not always explicitly or consciously acknowledged, there has always been a significant cost penalty associated with communication between physically separated circuit elements. Physical characteristics of the real world, such as resistance, inductive and capacitive coupling, the finite propagation speed of electrical impulses, the three dimensional property of space, etc., all conspire to favor communication between close rather than widely separated components.

Digital designers have, therefore, always sought to find logical groupings of circuit elements where the vast majority of communication is within the group and communication between groups is minimized. This is not simply an artifact of integrated circuit technology [Sutherland and Mead, 77]. For the full range of digital circuit technology--vacuum tubes, discrete transistors, and small, medium and large scale integration--we find a modularization of function. Circuit elements are grouped to form logic gates, flip flops, registers, adders, counters, arithmetic units, etc.

In digital design, the modular structuring of circuit elements leads to a decrease in cost and increase in performance because the costs of communication are reduced. These costs include time delay, power dissipation, pin utilization, etc. There are other important factors which encourage modular structuring. Modules need only be designed once and can be used repeatedly. This reduces costs through faster system design, faster production, reduced inventories and simplified maintenance.

With present technology, there are enormous economies of scale with integrated circuits. The development cost of an IC, such as the Intel 8080, may be several million dollars; its

production cost is close to three dollars.[1] Thus the ratio of development to production costs can be on the order of 10^6. This is a much higher ratio than with earlier modules based on printed circuit boards. This high ratio further increases the desirability of developing a small number of very flexible module types; such designs allow the largest production runs and hence the easiest amortization of development costs.

Another intriguing aspect of the economics of hardware design with integrated circuits is that, within some tight but expanding constraints, complexity within a chip is almost free. For example, if there are two closely communicating functional units which can be combined into a single composite function, on a single IC, then the marginal cost of the overall function may be close to halved[2]. It follows from this that there is a strong economic motivation for designers to find consistent functional blocks which will closely match the current optimum IC complexity. Successful examples of this are large memory chips and microprocessors. While there are other highly complex chips, these are usually rather specialized and have a limited market. Few other general purpose functional units exploit more than a small fraction of the potential complexity of a chip. Present day mini and large-scale computers are constructed with relatively low complexity ICs (except for memories).

In summary, a hardware designer is concerned with two complementary activities:

1. The discovery and definition of functional entities suitable for optimum utilization of available technology. The implementation of these modules, on a printed circuit board or within an IC, involves both the recursive application of this activity and classical gate level optimization techniques.

2. Functions which are complex, or are to be used only in small quantities, must be decomposed to utilize a minimum number of an appropriate set of less complex standard modules.

The overall hardware cost of a system is determined much more by the number of modules, and module types, than it is by the intrinsic (active component level) complexity of the system. This is a very strong motivation for computer designers to explore structures which allow simple decomposition into a small variety of appropriate sized modules.

1.4 Performance Improvement Techniques

It is well understood that all computers are equivalent in the sense that they can all be

---

[1] In this respect, the IC industry is similar to the software industry. Development costs are high but replication costs are low.

[2] This is subject to restrictions on power, pins and maintenance of acceptable yields with the increased die size.

used to compute any computable function (within the limitations of their memory). One dimension along which they differ is performance - the speed at which a given function can be computed. John von Neumann, in discussing instruction set design in 1946, expressed it this way:

> "It is easy to see by formal-logical methods that there exist codes that are in *abstracto* adequate to control and cause the execution of any sequence of operations which are individually available in the machine and which are, in their entirety, conceivable by the problem planner. The **really decisive considerations** from the present point of view, in selecting a code, are more of a practical nature: simplicity of the equipment demanded by the code, and clarity of its application to the actually important problems together with the speed of handling of these problems."[1]

In the period since this landmark paper, most performance improvements have come through the technological development of components. The development by physicists, chemists and engineers has been driven by strong military and commercial forces. However, this quotation does point out one of the prime ways in which computer designers have contributed directly to improving performance. This is in the selective optimization of frequently occurring operations. A second fundamental way in which performance has been improved is through concurrent execution of independent operations.

1.4.1 Exploitation of Skews in Frequency Distributions

A prime example of the exploitation of skews is the effective use of memory with an extremely wide range of performance characteristics. The few, most frequently accessed operands are stored in the fastest available memory. In a conventional processor, these are the machine registers. Less frequently referenced operands are stored in primary memory, which might be ten times slower. The same principle is extended to secondary and tertiary storage. If no account was taken of the distribution of references to operands and all operands were held in memory with uniform speed, then the system would either be much more expensive or much slower, probably both. Caches, or look-aside buffers, are provided for the same reason as registers. However, the programmer or compiler is relieved of the burden of statically identifying frequently referenced operands (or code loops) and allocating registers.

Stack-oriented architectures, string manipulation instructions, implementation of floating point in hardware, microcoding of operating system primitives, etc. all represent the selective optimization of operations which are frequently performed. Much of the skill of a

---

[1] "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument," Burks, Goldstein and Von Neumann, 1946. Emphasis added.

designer lies in defining primitives (for example, instructions) which will allow the direct representation of frequently required operations.

1.4.2 Parallelism and Concurrency in Computer Structures

The second important way of improving performance is through various forms of concurrency. A simple example is the overlapping of the execution of one instruction with the fetching of the next. This increases the utilization of both the processor and the memory system. Although this form of concurrency may increase control complexity, it need not significantly increase hardware costs.

With any specific implementation technology, there is a fundamental limit to the speed of operation. This speed limitation can only be overcome through additional hardware and using parallelism among the hardware units to increase overall system performance. The principle forms of parallelism in computer systems are:

1. Internal pipelining and look-ahead to exploit parallelism contained within a conventional instruction stream.

2. Multiple data streams, exploiting the parallelism inherent in algorithms expressible in vector or array form.

3. Multiple instruction streams, each with their own data stream, to exploit parallelism in asynchronous parallel algorithms and parallelism due to independent users.

The first technique, pipelining of instruction execution, is widely used but performance is strictly limited by the delay through is the slowest functional unit in the chain[1]. Thus there is a strict upper bound in performance, for any given implementation technology. The pipeline also depends on being able to anticipate program execution at conditional branches. The main advantage of pipelining is that it is completely transparent to the programmer. Manufacturers can have a family of computers with identical architecture, but vary the performance according to the degree of pipelining employed. Processor design for a pipelined machine is highly complex and is not amenable to a reasonable modular structuring. Unless some special reliability techniques are used, at additional cost, the processor is likely to be susceptible to any single failure.

Single Instruction, Multiple Data stream (SIMD) computers are restricted to algorithms expressible in a vector or array form. This includes some problems with large economic

---

[1] The performance of look-ahead schemes with multiple functional units is limited by sequential dependencies in the data stream.

significance, for example, processing seismic data and weather forecasting. The independent processing elements provide a natural modular decomposition. However, instructions are normally broadcast from a central control element: the consequent need to maintain close synchrony between Processing Elements (PEs) may limit both the speed and number of PEs. Reliability can only be provided via standby spares because vector operations are not tolerant of a variable number of processing elements. The narrow range of applicability limits the market for these machines, and so they do not benefit from economies of scale.

The third form of parallelism, with multiple independent instruction and data streams, has the potential of offering performance improvement for a wide range of applications. Small multiprocessors systems, with say two to four processors, have been used commercially for some time. Normally these are used to exploit parallelism between independent users in a timesharing facility. Pluribus [Heart et al, 73] is an example of a medium sized multiprocessor (the largest known system has 14 processors) which has been used for a single, dedicated task--a node in a packet switched network. C.mmp [Wulf and Bell, 72] is a fully operational 16 processor system with a sophisticated general purpose-operating system. A wide variety of tasks have been implemented and evaluated on C.mmp [Fuller and Oleinick, 76].

1.5 Multiprocessors, Networks and Vector Machines

The term multiprocessor is sometimes used to mean simply any collection of multiple computers, processing elements, processors, etc. which are capable of working cooperatively. In this document the term *multiprocessor* will be used uniformally to mean a collection of independent processors which communicate through shared access to primary memory. The term *network* will be used to mean a collection of independent computers (i.e., processor-memory pairs) which can communicate by the passing of messages. Computers in a network may be loosely coupled, as in the ARPAnet [Heart et al, 70], or cooperate more closely as in TANDEM [Tandem, 77]. Multiprocessors must also be distinguished from vector and array machines, such as ILLIAC IV [Bouknight et al, 72], which communicate via buffer registers in a fully synchronous, pre-programmed manner. Vector and array machines may have multiple processing elements but there is only a single point of control.

1.6 Applications of Multiprocessors

There exists little experience in applying multiprocessors with more than 2 to 4 processors. Pluribus [Heart et al, 73] has been used successfully as a node in a packet-switched network. C.mmp has been used for AI tasks, such as speech and vision, and various numerical applications. C.mmp, with the operating system Hydra, offers many of the facilities of a general purpose timesharing system. Systems with a small number of

processors are regularly used to augment the performance and improve the reliability of mainframe systems.

Use of a multiprocessor (except to merely improve reliability) requires the exploitation of concurrency within the application. This parallelism can found in broadly three ways:

1. **Multiple, independent users.** This may be separate programs run by different users, as in a conventional timesharing system. Another possibility is independent enquires in a large data base. Examples include airline reservation systems, credit verification and banking.

2. **Automatically extracted concurrency.** A potential for concurrency exists within most programs written for execution on a uniprocessor. However, the degree of parallelism possible is usually limited (in the range 2 to 10) and the cost of extracting the parallelism may be substantial [Hibbard et al, 78].

3. **Manually decomposed applications.** Many computer applications can be explicitly decomposed by a programmer to make effective use of a multiprocessor. Parallel algorithms exist for many standard numerical operations. Important commercial applications, such as sorting, can also be decomposed. There is a wide class of applications which appear inherently parallel and must be explicitly serialized for execution on a uniprocessor. Many real-time control tasks fall in this class because independent physical devices inherently operate in parallel. Many large Artificial Intelligence (AI) tasks, including speech and image understanding, have natural parallel decompositions.

Parallelism, or concurrency, can be found in the greatest abundance through explicit task decomposition. However, this may require substantial additional effort by the applications programmer. Hence the software costs associated with the application may be increased. Independent of software costs, it may be essential to use a multiprocessor to achieve real-time performance goals. This is the case with weather forcasting and on-line data-base enquires, for example.

There are several ways in which a multiprocessor might be substantially faster than the fastest available uniprocessors:

1. The multiprocessor may be composed of a modest number of processors which individually are close to the maximum performance possible with current technology. This is the case with the S1, which will have up to 16 high performance processors [McWilliams et al, 77].

2. A multiprocessor may utilize inexpensive, low performance processors. However, there may be a sufficient number to reach any desired total performance level. (This is the goal of Cm*.)

3. The potential for high I/O bandwidth in some multiprocessor structures may lead to higher performance on I/O limited tasks.

For non-time critical tasks, the software cost of explicitly decomposing an application may only be justified if the multiprocessor offers substantially better cost-performance than a uniprocessor. A cost-performance analysis of a multiprocessor is extremely complex. Sam Fuller has attempted to compare a PDP-10 with C.mmp [Fuller, 76]. A full analysis would have to include a measure of the additional software cost in using a multiprocessor, a measure of relative expenditures on hardware and software, and a measure of the expected processor utilization.

It may be that the perceived programming difficulties of multiprocessor algorithms are due to a lack of experience, unhelpful system architectures and poor software tools. If the programming difficulties of multiprocessors can be overcome, then the potential for extensibility, reliability and cost-performance will make them very attractive.

1.7 Multiprocessor Switching Structures

Figure 1-1 shows the conceptual structure of all multiprocessors. There is a set of processors, $P_1 \ldots P_p$, which have access to a set of memory modules, $M_1 \ldots M_m$, via a switch, S. (This diagram, and many later in the thesis are in PMS notation [Bell and Newell, 71].) The instruction set and other characteristics of the processors are very important, just as in uniprocessor systems. However, most of the design considerations which apply to processors for a multiprocessor also apply to uniprocessors. The memory modules for a multiprocessor may be identical to those used in uniprocessors. It is the switching structure, giving processors access to shared memory, which distinguishes multiprocessors from uniprocessors and provides the great wealth of possible multiprocessor structures.

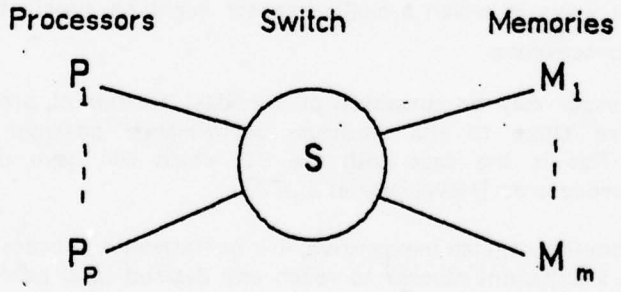


Figure 1-1: Abstract Multiprocessor

The cost and performance of some of these structures will be discussed in this section. We will be particularly concerned with the properties of extensibility.

1.7.1 Switch Characteristics

We are considering the switching structure of a fully asynchronous multiprocessor. From the viewpoint of an individual processor the switch structure imposes a delay which is additive with, and indistinguishable from, the access time of shared memory. The effective access time of an arbitrary reference through the switch to shared memory is the only characteristic which directly effects processor performance.

There are two important switch characteristics which have a strong bearing on the effective access delay seen by a processor:

1. **Contention-free Switch Delay** is the delay time through the switch under conditions of no contention for either the target memory or any data path within the switch. This gives a lower bound on the effective access time to shared memory.

2. **Switch concurrency** is the average number of concurrent references the switch structure will support without significantly increasing the switch delay. This assumes no memory contention.

1.7.2 Full Concurrency Switch Structures

In most present-day multiprocessor structures, the cost of shared primary memory is substantially larger than the cost of any reasonable switching structure to access that memory[1]. However, the delays imposed by the switch directly detract from the effective memory performance. Therefore designers normally opt for maximizing switch performance in order to maximize utilization of shared memory. Maximum switch performance corresponds to a uniform structure with minimum Switch Delay and full Concurrency:

1. In practice, minimum switch delay is determined by the implementation technology, arbitration circuitry, cable lengths, etc. However, we need not consider these important engineering factors here; instead, we will note that a path, from a processor to a memory, through a switch can utilize one or more switching levels. We will see below that multilevel switches offer substantial cost savings (particularly for large systems) but impose a time penalty for each

---

[1] While this may be true on an overall cost basis it is not so clear on an Integrated Circuit count basis. For example, the C.mmp central crosspoint switch uses approximately 5000 (16 pin equivalent) Medium Scale Integration ICs. C.mmp at present has 4 million bytes of memory. With 1973 technology this is about 8,000 4K RAM chips, with 1977 technology it is only 2,000 16 K RAM chips. Technology is rapidly decreasing the per bit price of memory but the cost of the switch structure, because it is pin count and power limited rather than complexity limited, is only slowly decreasing. Thus for multiprocessors with 8 or 16 processors and a crosspoint switch, the switch cost may soon dominate the overall system cost.

additional switching level. Therefore a switch with minimum delay will have only a single switching level.

2. A full concurrency switch is one which allows, for some reference patterns, all processors to concurrently access memory without additional delay due to contention for data paths in the switch. In conventional systems, with uniform access to shared memory, a full concurrency switch will allow all 1 to 1 mappings between processors and memories without contention.

We have informally established that a switch structure with maximum performance will have one switching level between processor and memory, and that an arbitrary 1 to 1 mapping between processors and memory can be established. There is only one switch structure with these properties. It is the crosspoint switch. This switch may be implemented in a fully centralized way; this is shown in Figure 1-2. The prime example of this is C.mmp which has a 16x16 centralized, crosspoint switch.

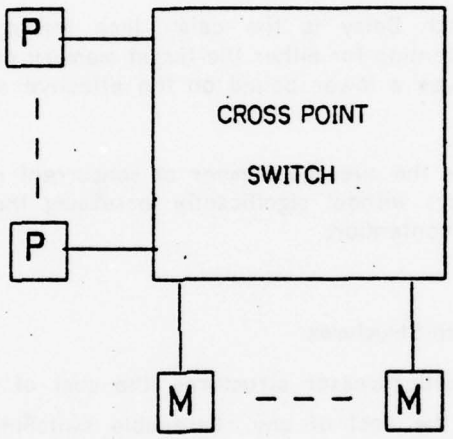


Figure 1-2: A Multiprocessor with a Centralized, Crosspoint Switch

A much more common implementation is to partition the switch and physically associate sections of the switch with corresponding memory modules, as shown in Figure 1-3. This is the multiported memory structure used for traditional multiprocessors such as IBM System 370, DEC System 10, etc. The principal reason for the popularity of this physical structuring is that many single processor systems require multiported memory for high bandwidth I/O. It also allows more flexibility in the system configuration, subject to the limit of the number of processors being equal to the maximum number of ports in the memories.

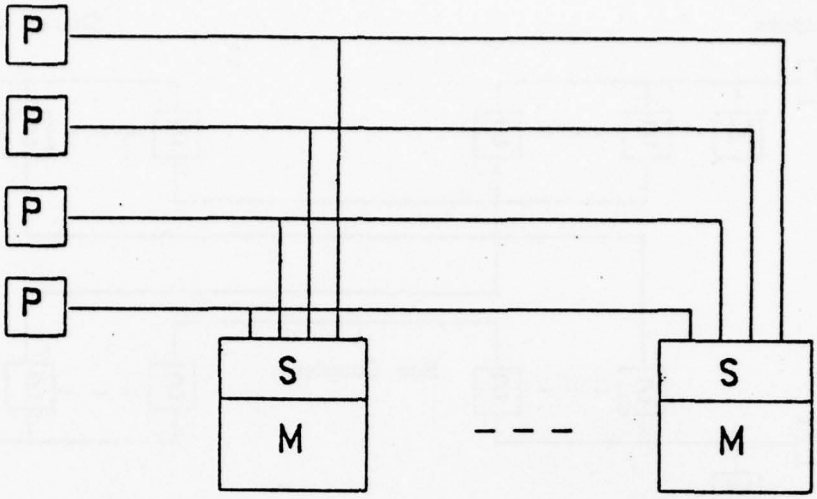


Figure 1-3: A Multiprocessor Using Multiported Memory Modules

A centralized crosspoint requires fewer cables and probably is faster (for the same technology) than using multiported memories. However, any failure is likely to be more catastrophic than in the distributed system and maintenance probably requires bringing down the entire computer system.

A crosspoint switch can be further partitioned. In Pluribus access to shared memory and shared I/O devices is via "bus couplers". A "bus coupler" is essentially a modularized, single crosspoint. A Pluribus system consists of a number of processor buses (actually supporting two processors and some local, private memory) and a number of shared memory buses. (See Figure 1-4.) A "bus coupler" is connected between each processor bus and each shared memory bus. A significant performance penalty is imposed by the delays through the bus coupler and arbitration on the shared memory bus. This degradation is offset by ensuring that most references are to the private memory, local to each processor. The bus couplers may also be a major cost factor. For 16 processor buses and 16 memory buses, 256 bus couplers and associated cables are required. The prime objectives in the Pluribus system were reliability and modular expansion, rather than high performance or low cost.

1.7.3 The Cost of Full Concurrency Switches

In all the implementations of full crosspoint switches, the total number of switch elements

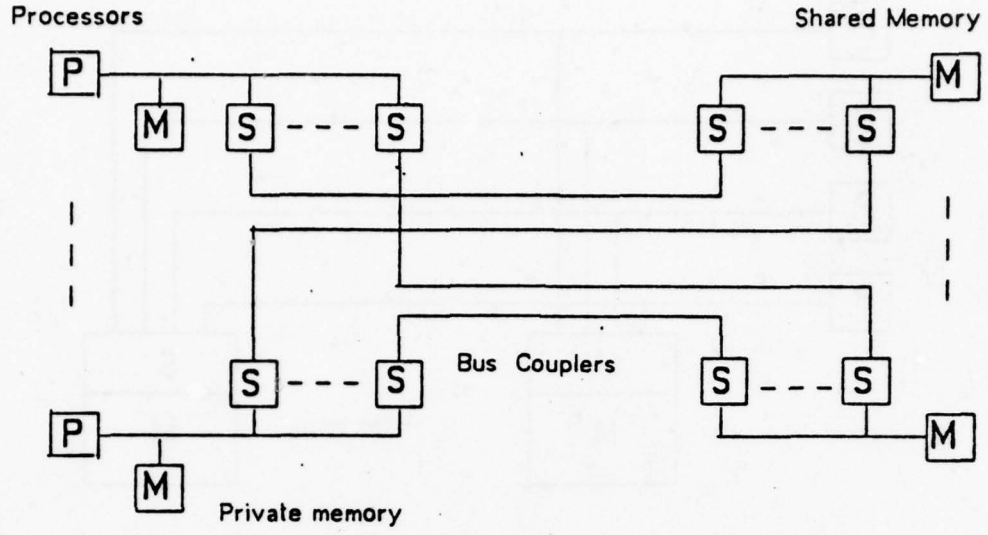


Figure 1-4: A Multiprocessor with a Distributed, Crosspoint Switch

(each switching a full data path) is proportional to p*m (where p is the number of processors and m is the number of memories). Thus, for normal configurations where p = m, the cost of the switch grows as the square of the number of processors.

The C.mmp central switch, designed in 1973, probably represents close to a minimum cost implementation for a high performance, full concurrency crosspoint switch using the best commercially available technology of the time. Any partitioning or distribution of the switch would have decreased performance and increased costs. Figure 1-2 gives a tabulation of the cost, in 16 pin chip equivalents, of various parts of C.mmp. The basic switch cost is almost 5000 chips. This is for a 16x16 crosspoint switch with 16 bit data paths and 24 bit address paths. The control portion contributes approximately 30% to this cost. A pilot, 4*4 version of the C.mmp switch required approximately 700 chips. The less than quadratic cost growth is due to a less than optimum utilization of components in the experimental 4*4 scheme.

Even with a centralized and highly optimized design, switch delay can greatly affect overall performance. For example, C.mmp incurs a minimum of 250 ns delay[1] in the switch when accessing memory which has a 250 ns cycle time. For the PDP-11/40 processors used on C.mmp, which can reference memory about every 1000 ns, this amounts to a 25% to 33% overhead.

---

[1] The switch is currently operating with a 400 ns delay. It is believed that it could be tuned to 250 ns.

Table 1-2: Chip Usage Analysis of C.mmp

	Unit	System Size 4 * 4	16 * 16
Processors			
Processor (PDP-11/40)			416
CMU Modifications			56
Relocation Register (Reloc 1 + Reloc 2)			103
Master Clock (1 per system)			83
Local Clock (1 per processor) (Also used for interprocessor interrupts)	116	464	1856
	774	2,847	11,139
Switch, excluding memory			
Processor Interface	25	100	400
Memory Priority Decoder + High Priority Driver	54	216	864
Memory Control	28	112	448
Card for 4 * 4 Data paths (7 needed)	43 * 7 =		301
Card for 16 * 16 Data paths (70 needed) (8 * 16 pin, 16 * 24 pin)	37 * 70 =		2590
Crosspoint Enable	34	136	544
		865	4846
Processor plus Switch		3,712	15,885
Memory (C.mmp has a mixture or core and semiconductor memory, this tabulation assumes all semiconductor with 4K RAM chips.)			
512 K (18 bits, non RAM = 1,188, RAM = 2304)		3492	
1,280 K (18 bits, non RAM = 2,970, RAM = 5,760)			8730
Grand Total (16 pin equivalents)		7,204	24,615

1.7.4 Private Memory and Caches

Many multiprocessors provide each processor with a significant amount of local, private memory. One important reason for this is to reduce the number of references to shared memory, through the main switch, and hence reduce the switch bandwidth requirements. In Pluribus this is done by statically, duplicating frequently referenced code in memory private to each processor. In the S1 [McWilliams et al, 77] this is done in the form of a cache. Thus, code is dynamically migrated from shared memory to cache memory, private to the processor. Note that for the purpose of reducing switch traffic, the cache memory need be no faster than standard primary memory. (However, in the S1, the cache memory is faster than main memory and so serves to increase processor performance.)

Caches are not suitable for shared, writable data[1] in a multiprocessor . This is because the cache of one processor can continue to hold the old value of an operand after is has been updated by another processor. Caches are best suited for read-only data, particularly code. However, writable data which is guaranteed to be private to single process, for example a stack segment, may also be cached successfully. However, the cache contents may represent a substantial amount of process state which must be transferred to shared memory before a process can be dispatched on another processor.

The provision of explicitly addressed private memory (i.e., not a cache) is somewhat antithetical to the notion of processors cooperating through shared access to memory in a multiprocessor. From an operating system viewpoint it introduces a resource which must be allocated independently from sharable primary memory and which can only be used in restricted ways. Any use of memory private to a particular processor will bind a process to that processor. This may be an embarrassment if the processor fails or is heavily loaded while others are idle. There is also a problem with I/O. If the memory is only accessible by a single processor it is unlikely to be accessible by I/O devices elsewhere in the structure. Thus all data transfers must be done by program loops executed by the local processor.

1.8 Non-Crosspoint Switches

Most multiprocessors are based on a crosspoint switch, either centralized or distributed, which provides uniform access time to shared memory. The prime drawback of this approach is the cost. This cost becomes prohibitive for multiprocessors with more than 16 or 32 processors. We will examine some less costly switch structures.

---

[1] Schemes for correctly invalidating or updating all caches holding any datum overwritten by any write operation have been proposed. However, implementation of these schemes requires a very high bandwidth communication path connecting all caches, and this is infeasible in high performance systems.

1.8.1 Low Concurrency Switches

The cheapest possible switch structure is to allow all processors and memories to share a common bus. This is shown in Figure 1-5.

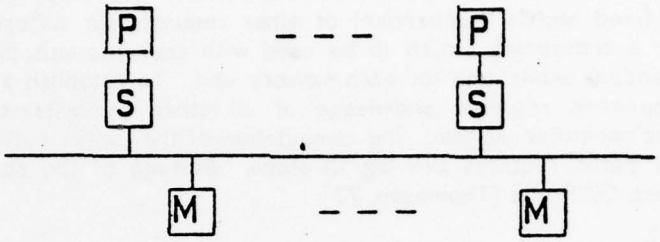


Figure 1-5: A Low-Concurrency Switch

This switching structure is simply a limiting case of a crosspoint switch. Switch cost is proportional to p*m, where p is the number of processors and m = 1. While there may be multiple memory modules, there is only one data path to shared memory and hence logically only one memory unit. This emphasizes the obvious limitation of the structure: overall performance is limited by the bandwidth of a single bus. Methods for reducing contention for this critical resource include provision of private memory, local to each processor, and caches. (See Section 1.7.4).

1.8.2 Connection Networks for Vector Machines

A connection network is a switching network with N inputs and N outputs, capable of performing any 1-to-1 mapping of input to output [Benes, 65; Thompson, 77]. A crosspoint switch is one example of a connection network. An active research area is finding the minimum number of switching elements necessary. The best known connection network takes $4\ N\ \mathrm{Log}_2\ N$ switch elements and has a delay of $2\ \mathrm{Log}_2\ N - 1$ switch stages[Benes, 65]. This is substantially cheaper than the N^2 switch elements in a crosspoint switch. These switch structures have been proposed for use in vector machines, like Illiac IV, to allow interchange of operands between synchronous processing elements. For this application the multiple levels of the switch need not be physically realized, they represent steps in an algorithm to route data. In this work, we are not concerned with the design of vector machines. These switch structures have two major disadvantages which make them unsuitable for a

multiprocessor:

1. For realistic size switch structures the switch delay, which is uniform for all accesses, would substantially degrade performance. For 16 processors, there would be 7 levels of switching delay rather than the 1 level in C.mmp. For 512 processors there would be 17 stages of delay.

2. In a crosspoint switch, the path from a particular processor to a particular memory is fixed and is independent of other requests for different memories. This allows a crosspoint switch to be used with asynchronous processors and with independent arbitration for each memory unit. To establish a path through a Benes network requires knowledge of all other requests--this implies a synchronous computer system. The computation of the switch settings to enable the correct paths requires O(N log N) steps. Storage of the correct settings would require O(N!) bits [Thompson, 77].

1.9 The Derivation of the Cm* Switching Structure

The switch structures discussed above all have the property of presenting a uniform delay on access to memory. In Section 1.4.1, we discussed the numerous ways to exploit skews in the frequency of some operations. In this section we will explore how another skew, locality in memory reference patterns, can be exploited in a switching structure.

We will make two assumptions:

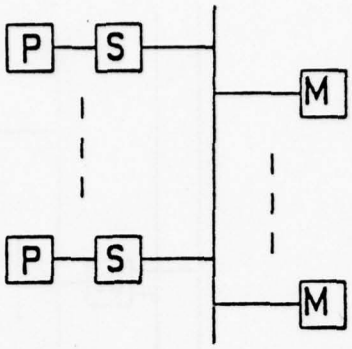
1. All processors must be able to access all memory. I.e., the structure will be a true multiprocessor without any private memory.

2. For each processor there will be a designated memory. It will contain sufficient code and private or read-only data to ensure that most references are to this designated memory.

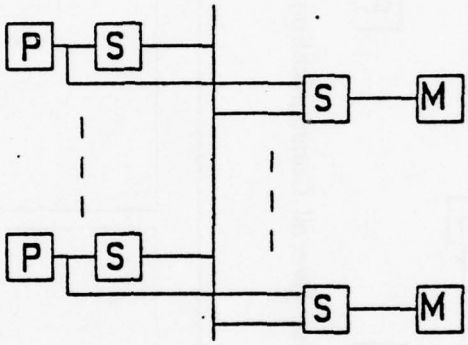
Figure 1-6(a) shows a simple structure which satisfies the first assumption. It is a single bus, low concurrency switch. To reduce contention for this single bus, a special data path is added from each processor to its designated memory (Figure 1-6(b)). Thus the switch structure has a high concurrency for references by processors to their designated memories. References to other than the designated memory must contend for the low concurrency data path.

1.9.1 Local-Memory and Shared-Memory Duality

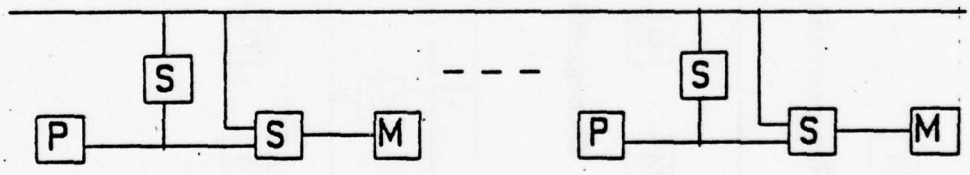
The representation of this structure can be manipulated into a more familiar form. Figure 1-6(c) shows how the designated memory may be viewed as being local to a particular processor. (A further simplification of the representation is shown in Figure 1-7(a).) This local memory is also the shared memory in the structure. This local-memory and



(a) All Processors have Access to All Memory



(b) Add Private Data Path to Designated Memory



(c) Rearrange the Representation of the Structure

Figure 1-6: Derivation of the Cm* Switching Structure (Part 1)

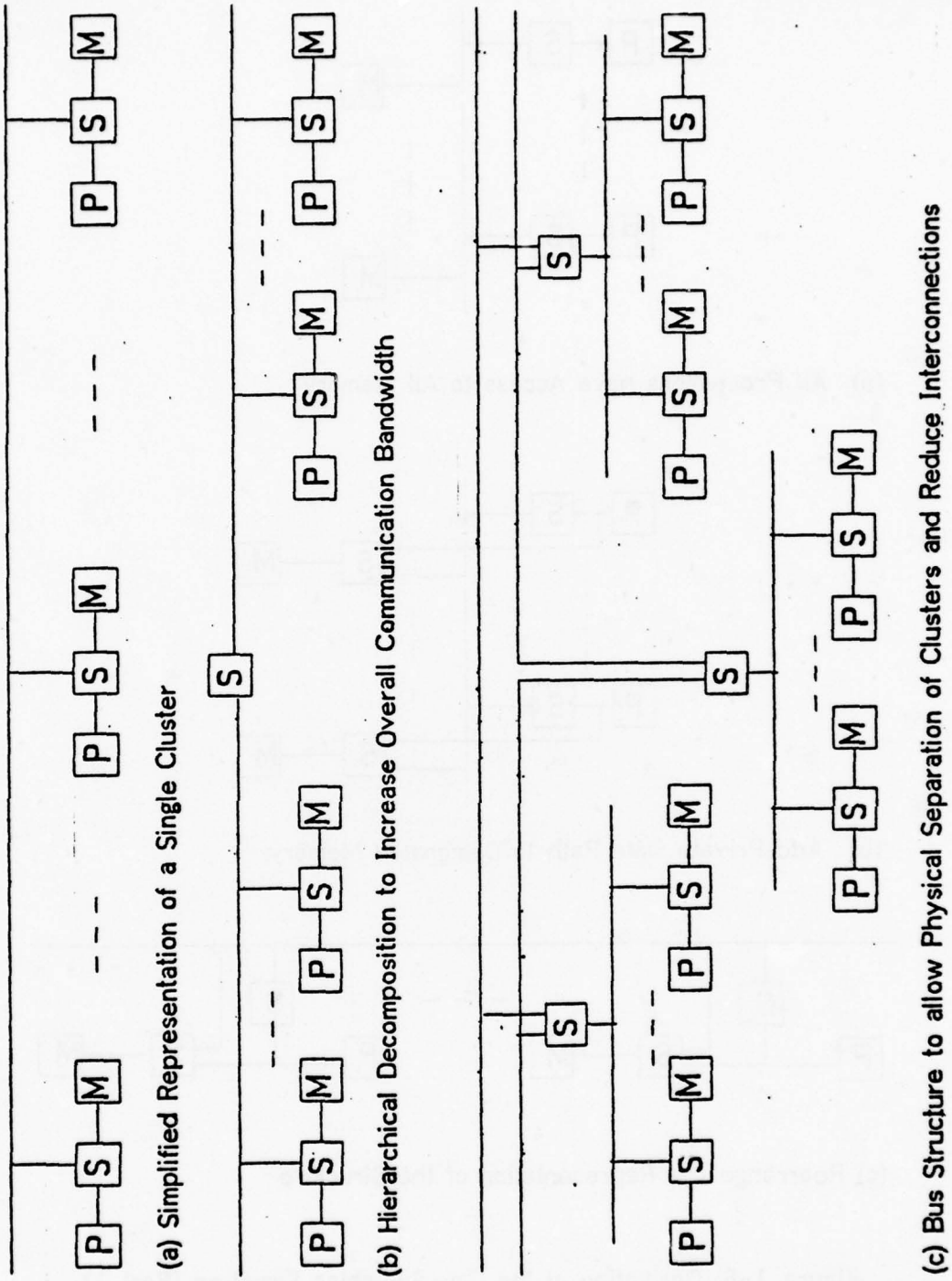


Figure 1-7: Derivation of the Cm* Switching Structure (Part 2)

shared-memory duality is the single most important characteristic of the Cm* structure.

The single bus (Figure 1-7(a)) for the low concurrency path will still limit expansion of this structure. References on this bus represent communication between the processors. A single bus would imply a limitation on communication bandwidth. To avoid this limitation, we apply the locality argument again. The single bus can be divided into two or more independent parts (clusters) with a low concurrency path between them. This is shown in Figure 1-7(b). For flexibility, reliability, and to avoid limitations on communication between "clusters", a network of buses can be used to to interconnect clusters of processor-memory pairs (Figure 1-7(c).

We have derived a structure very close to the actual hierarchical structure of Cm*. The number of switching elements is approximately 2*p (where p = m). This is only twice the switching structure cost of a simple, low concurrency structure. However, provided the locality argument applies, the structure has close to the effective concurrency of a full crosspoint switch which costs $O(p^2)$.

We will see in subsequent chapters that implementing such a structure is not as simple as this brief introduction might suggest. However, it has been done--at least on a small scale. Experience with a 10 processor Cm* system indicates that relatively little effort is required to achieve the program locality (for a limited range of tasks), and that performance is comparable to a system with a full crosspoint switch.

1.10 Communication, Cooperation and Architecture

When multiprocessors are used to simply exploit parallelism between independent users the demands on the operating system (O.S.) and architecture are little different from a uniprocessor's. Certainly, the O. S. needs to be reentrant in some form, and mutual exclusion locking on internal data structures is required. However, the frequency and nature of the operating system calls have not changed.

When a multiprocessor is used to exploit concurrency within a single user's application, additional demands are made on both the O. S. and the architecture. The exploitation of concurrency, to achieve performance improvement, requires cooperation between asynchronously executing processors. Cooperation requires communication. The ease and efficiency of this communication dictates the grain size of concurrent computations which can be effectively exploited [Oleinick and Fuller, 78]. Oleinick and Fuller give measured results, from applications on C.mmp under Hydra, where use of high overhead communication primitives can lead to severe performance degradation as additional processors are used.

To a large extent, the achievement of efficient communication and cooperation between processors is the single major theme of this thesis. We have already discussed the mechanisms, at a hardware structure level, required to implement memory sharing. This is the basic primitive for communication in a multiprocessor. In Chapters 2, 3 and 4 this structural level of communication will be explored. This discussion will also include basic control primitives such as locks and interprocessor interrupts. Much of this work will be in the form of a derivation and examination of the structure of Cm*.

The physical structure required for communication is only half the story. The second half is the logical structure, or architecture, which allows operating systems and applications programs to exploit the structure. In Chapter 5, a simple model is presented which allows systematic discussion and comparison of addressing structures on a uniform basis. Chapter 6 is an examination of the addressing architecture of Cm*.

2. Cm*: Objectives, Constraints and Overview

The individual technical issues in the design and implementation of a large computer system like Cm* cannot be understood without an appreciation of the goals of the project. The first part of this chapter outlines these goals. The historical background of the Cm* project is also discussed. The second part of this chapter is devoted to an overall description of Cm* to provide background for the remainder of the thesis. This is followed by a brief survey of some of the development techniques used and a report on the project status.

2.1 Objectives for the Cm* Structure and Architecture

The main objectives in the design of the Cm* structure and architecture are to:

1. Provide a multiprocessor structure which allows the indefinite expansion, at close to linear cost, of:
   - Processing power.
   - Memory capacity.
   - Overall communication bandwidth.
2. To complement the physical structure, provide architectural support for efficient, close and protected communication between processes.
3. Provide an experimental test bed for a variety of architectures and special purpose scheduling and communication primitives.
4. Provide a system with no critical central resources, which can be dynamically reconfigured in the event of failure for fail-soft operation.

2.2 Cm* Viewed as an Extensible Multiprocessor

With the objectives given above in mind, we can examine the Cm* structure (Figure 2-1). Every processor in the structure can access every memory word. Hence it is a multiprocessor in the sense defined in Section 1.5. Processing power can be expanded by increasing the number of Cm's per cluster or by adding additional clusters of Cm's. Memory capacity can be increased by either adding it to an existing Cm or by adding additional Cm's.

The communication bandwidth of an individual processor in Cm* is limited both by its own performance and the bandwidth of the map bus and the Intercluster buses. Because there is no central bus or switching mechanism and the system can be indefinitely[1] extended there is

[1] In the present implementation of Cm* there is an effective limit to the total number of clusters because of address limitations (a maximum of 2^{28} bytes, defined in microcode) and considerations of deadlock in the intercluster buses. The system can support more than 10,000 processors without deadlock. See Section 3.6.

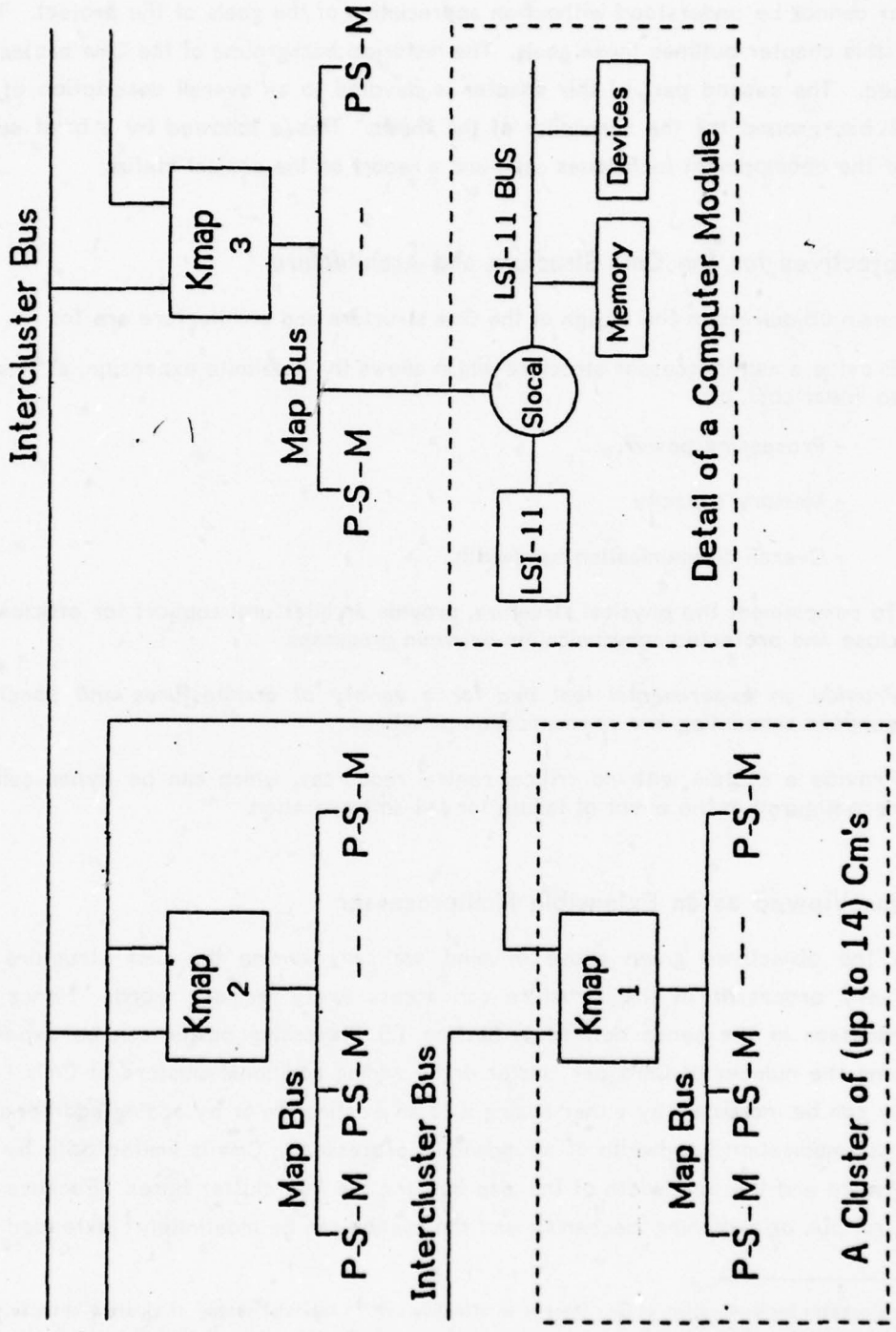


Figure 2-1: A Simple, 3 Cluster Cm* System

no limit to the total inter-processor communication. Provided an application can be suitably decomposed, any limitation on inter-processor communication due to saturation of a map bus can be solved by redistributing the Cm's with a larger number of clusters. Similar considerations apply for Intercluster bus saturation.

2.2.1 Hierarchic Communication and Application Decomposition

Cm* is a *hierarchic system* in the sense used by H. A. Simon [Simon, 62].

> ... a system that is composed of interrelated subsystems [clusters and groups of clusters], each of the latter being, in turn, hierarchic in structure until we reach some lowest level of elementary subsystem [Cm].

In Simon's usage, and in Cm*, there is no implied authority relationship between components of the hierarchy.

A consequence of the hierarchical physical structure of Cm* is the need to hierarchically structure communication within applications. The switching structure imposes a time penalty on memory references proportionate to the number of levels in the switching structure hierarchy traversed. At the bottom level of the hierarchy, a processor referencing its own local memory incurs no switching delay. At the next level, a reference within a cluster is delayed by a factor of 3. References between clusters with a common intercluster bus are delayed by a factor of 9. Thus to achieve good processor utilization, the number of non-local memory references must be limited. Broadly, the percentage of references which traverse a specific number of switching levels should, at most, be in inverse proportion to the delay incurred by those references.

Experience to date indicates that it is relatively easy to achieve sufficient locality, in a single cluster system, with a range of applications. (There has been no experience with the decomposition of applications for multiple clusters.) Applications decomposed for C.mmp, a system with uniform access time to memory, have been transferred without restructuring to Cm*. The applications (P.D.E.'s, integer programming, speech recognition) loose only 5% to 20% due to the non uniform memory access time. This is a prime example of the principle discussed in Section 1.4.1; great cost savings have resulted from exploiting a skewed frequency distribution.

In applications decomposed for Cm* to-date, most locality has come from duplicating code in the local memory of each processor. In a PDP-11/LSI-11 approximately 70% of all references are to code. Locality is increased to 90% to 99% by ensuring that the stack and other private data is in local memory.

Effective use of Cm* systems with multiple clusters, and groups of clusters, will require

further levels of locality. The range of applications which can be decomposed in this way is an open research question. Promising areas include: large AI systems such as Hearsay II [Lesser et al, 75], real time control, program verification [Jefferson, 78], weather modeling, quantum chemistry calculations and large data bases. H. A. Simon gives examples of hierarchical systems from a wide range of fields. He asserts that, "Empirically, a large proportion of the complex systems we observe in nature exhibit hierarchic structure" [Simon, 62]. Simon's definition of hierarchic systems has a remarkably close match to the properties of Cm*.

> We have seen that hierarchies have the property of near decomposability. Intra-component linkages are generally stronger than inter-component linkages. This fact has the effect of separating the high-frequency dynamics of a hierarchy--involving the internal structure of the components--from the low-frequency dynamics--involving interaction among components.

H. S. Simon--The Architecture of Complexity, 1962.

2.2.2 Supporting Close Cooperation between Processes

A crucial concept in using a multiprocessor to gain performance advantages is communication and synchronization between cooperating processes. Primitives such as mailboxes, semaphores, process creation and control, etc. are not normally provided within the architecture of a processor; they must be implemented within the operating system. Oleinick and Fuller [1978] have shown that the performance of an algorithm can be crucially dependent on the overheads associated with particular synchronization primitives. There are two consequences of high overheads for frequently used primitives: 1) Where possible, programmers will use faster, less general, and less well structured mechanisms in order to avoid invoking mechanisms provided by the operating system. 2) Where impossible to avoid, the primitives may substantially degrade overall performance.

It was argued in Section 1.3.1 that in any general purpose computer system it was particularly important to provide architectural support for the operating system. This is partially based on the assumption that overheads in the operating system will be reflected back to user programs. It also acknowledges that the operating system is often the single largest and most complex piece of software developed for a machine. The need for an architecture which enhances programmability is particularly acute for a multiprocessor. Multiprocessors appear to be inherently more difficult to program than uniprocessors; this is probably partially due to a lack of experience with multiprocessors but concurrency and non-determinism undeniably add to software complexity.

A major attraction of multiprocessors is that high performance systems can be built by combining many, cheap processors. Almost inevitably, these cheap processors are mini or

micro processors originally designed for dedicated, non-operating system environment, applications. This is certainly true for Pluribus, C.mmp, and Cm*. Thus there is a need in the multiprocessor structure and architecture to compensate for deficiencies in the processing elements. Uniprocessors with a comparable applications domain, would have a basic processor architecture intended to support an operating system.

2.3 The Modular Construction of Large Digital Systems

Up to this point in our presentation, the Cm* structure has been developed as an extensible multiprocessor suitable for a wide range of computational tasks. However, historically, Cm* had its begining in the Register Transfer Module (RTM) project [Bell et al, 72] at CMU. RTMs are a module set for the systematic construction of digital systems. The successor to the RTM project, and immediate forebear of Cm*, was the Computer Module (CM) project [Bell et al, 72; Fuller et al, 73]. This too had as its prime objective the systematic construction of digital systems. The Computer Module project was based on two main assumptions.

The prime assumption of the Computer Module project was that a simple computer, a processor-memory pair, is an appropriate module for building large digital systems. A computer is a completely general purpose device which, within performance constraints, can perform any well defined digital function. A computer is also well suited to interfacing sensors, actuators, storage media, communication devices, etc. Many of the motivations for modular structures are given in Section 1.3.2 and will not be repeated here.

A second basic assumption was that communication between modules should be at the level of a single memory reference. An example of a lower level of communication would be to postulate a network of control lines which could be set and sensed by individual processors. Synchronization might be achieved by use of a central clock and data passed via communication registers. This very low level of inter-module communication was rejected because it would too tightly constrain the use of the system. Any need to synchronize modules to a central clock would limit the physical size of the structure and severely impair the ability to build fault-tolerant systems.

Alternatively, an example of a higher level of communication would be to pass messages between the computers. This leads to a computer network, for example the UCI-DCS [Farber and Larson, 72], HXDP[Jensen, 78], Tandem [Tandem, 77], and the ARPAnet. Even with very high bandwidth interconnections, the grain size of effective cooperation between computers is limited by the overheads of message preparation, reception and activation of the designated recipient process.

2.3.1 Computer Modules

In the period 1973 to 1975 a range of module designs were explored. The designs were based on the two assumptions above and were oriented towards taking advantage of progress in semiconductor technology. These designs are briefly reviewed here.

The "Canonical Computer Module" structure is shown in Figure 2-2. The basic module is a processor-memory pair with a special interface to a number of inter-module buses. A reference generated by the processor may be directed, according to the address, either to the memory local to the processor or to the memory of any module directly or indirectly accessible via the network of inter-module buses. The path of a reference from processor P_1 to memory M_4 is shown in Figure 2-2.

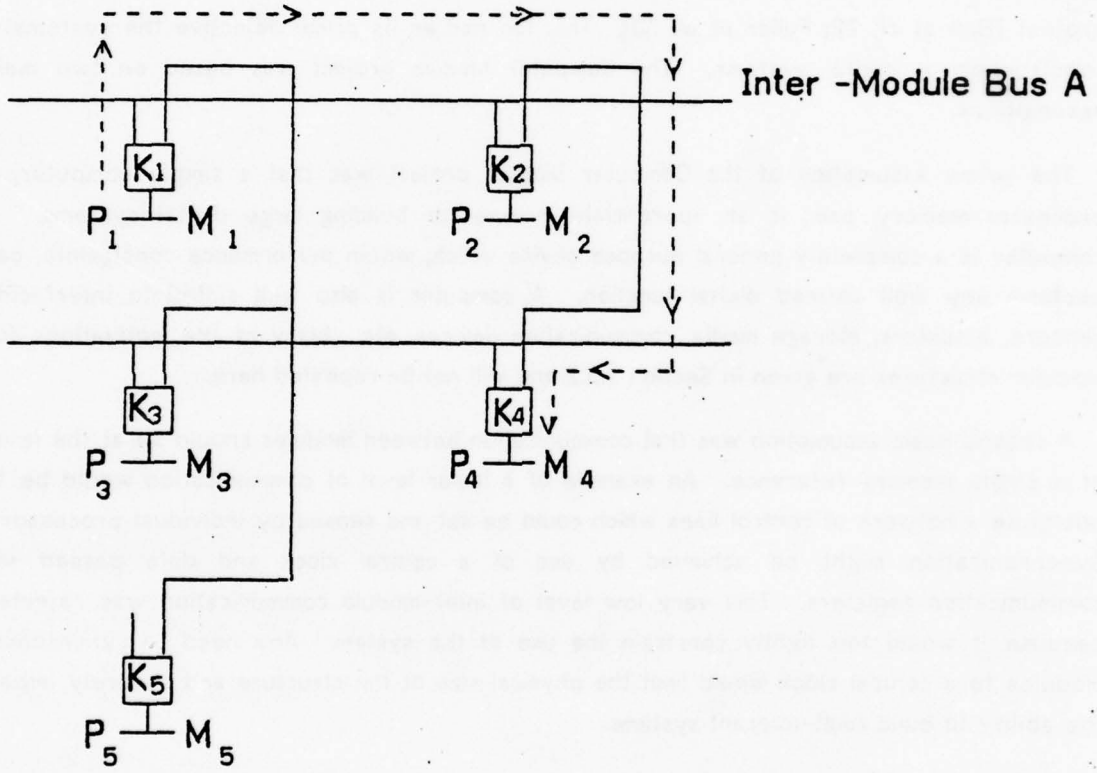


Figure 2-2: Canonical Computer Module Structure

Several alternate implementations of this general scheme were considered [Swan et al, 76]:

1. Direct implementation of the buses and address mapping logic in hardware for each processor-memory pair.

2. Use of simple point-to-point links for interconnection of modules. Address mapping and routing of requests through intermediate modules would be done in microcode by each processor.

3. Connect the processor-memory pairs into groups, called clusters, with a single bus. Most of the address mapping, routing and bus control functions are centralized for each cluster.

The cost of the first two options is dominated by the number of ICs required to interface to the buses or links. From the viewpoint of possible future implementation with LSI, both designs are deficient because of the high power dissipation for bus drivers and the high pin counts required. Evaluation of the delays in address mapping and routing in microcode indicated that the second option would be substantially slower than the other two. The third option is substantially cheaper (by a factor of 2 to 4 in chip counts for interconnections) and offers considerably more flexibility in the addressing mechanism. This is essentially the structure used for Cm*.

2.4 A Brief Description of Cm*

The structure of Cm* is described in reasonable detail elsewhere [Swan et al, 77a; Swan et al, 77b; Fuller et al, 78]. However, as an aid to the reader a description of the major components and an overview of the operation of Cm* will be presented in this section. The above references and Chapters 3 and 6 of this document should be consulted for further details. (Many of the figures in this section are reproduced from [Swan et al, 77b].)

2.4.1 The Components of Cm*

Figure 2-1 shows the primary components of the hierarchical Cm* structure:

- A **Computer Module** or **Cm** provides the processing power, primary memory and I/O connections for the system.

  - The **processor** is a Digital Equipment Corporation LSI-11. This is a 16 bit microcomputer with the same instruction set as the PDP-11 family. Its speed is about 0.14 Million Instructions per Second.

  - The **Slocal**, or local switch, is special purpose hardware built at CMU. Its principal function is to interface between the LSI-11 processor, the LSI-11 bus and the Map Bus. (The Map Bus connects the Cms in the cluster, see

below.) It also provides numerous other functions, including: user/kernel address spaces, multilevel interrupts for devices and interprocessor communication, detection of privileged instructions, parity logic, etc. (See Figure 2-3.)

- The memory units and devices on the LSI-11 bus are standard commercial products. The existing 10 processor system has 56K bytes of memory per Cm. Cm's in the 50 processor system will have 128K bytes each.

- The computer modules are combined into a cluster via the Map Bus. The Map Bus is a special purpose, time-multiplexed (or packet-switched) bus with a transaction time of about 500 ns. The Map Bus protocol is discussed in Section 4.3.
- The Kmap is a special purpose "mapping controller" which is shared by a cluster of Cm's. Clusters are connected via Inter-cluster Buses. All non-local memory references in Cm* are handled by one or more Kmaps. The Kmaps provide address expansion and mapping, and routing of references both within a cluster and between clusters. The Kmaps were designed and built at CMU. The components of the Kmap, shown in Figure 2-4, are:

  - The Kbus, or bus controller, provides the interface between the Map Bus and the Pmap and controls all transactions on the Map Bus. (Components of the Kbus are shown in Figure 2-5.) The Kbus arbitrates (with a pseudo round-robin discipline to avoid starvation[1]) all requests from Cm's on the Map Bus. It also performs context allocation for the Pmap and provides a time-out mechanism.

  - The Pmap or mapping processor is a horizontally microcoded, special purpose processor. (The Pmap data paths are shown in Figure 2-6.) It has eight independent sets of general purpose and subroutine return registers. This provides efficient hardware support for multiprogramming or multiplexing of the Pmap. Each set of registers is called a Context. The Pmap is specialized for the address mapping task by provision of field extraction logic, extensive multi-way branching facilities and 10K bytes of local, fast data storage. The Data RAM is arranged as 1K records each of five 16 bit words. This holds segment descriptors and other special data structures for address mapping. The (writable) control store is 4K words by 80 bits.

  - The Linc is the interface to the two inter-cluster buses. (The major components of the Linc are shown in Figure 2-7.) Under the direction of the Pmap, a message (up to eight words) can be transferred to any other Linc with a common bus. (Clusters not directly accessible via a common inter-cluster bus are reached by forwarding via intermediate Kmaps.) The Linc contains a buffer area for 128, eight word messages--a message is normally two, three or four words. The Linc maintains, in hardware, a system of queues of message buffer pointers. The objective of the design is to provide very fast packet transactions between clusters with a minimum of support from Pmap microcode. (Linc performance is discussed

[1] A Round-robin, rather than priority, discipline is necessary to prevent repeated requests from a group of higher priority Cm's blocking requests from lower priority Cm's.

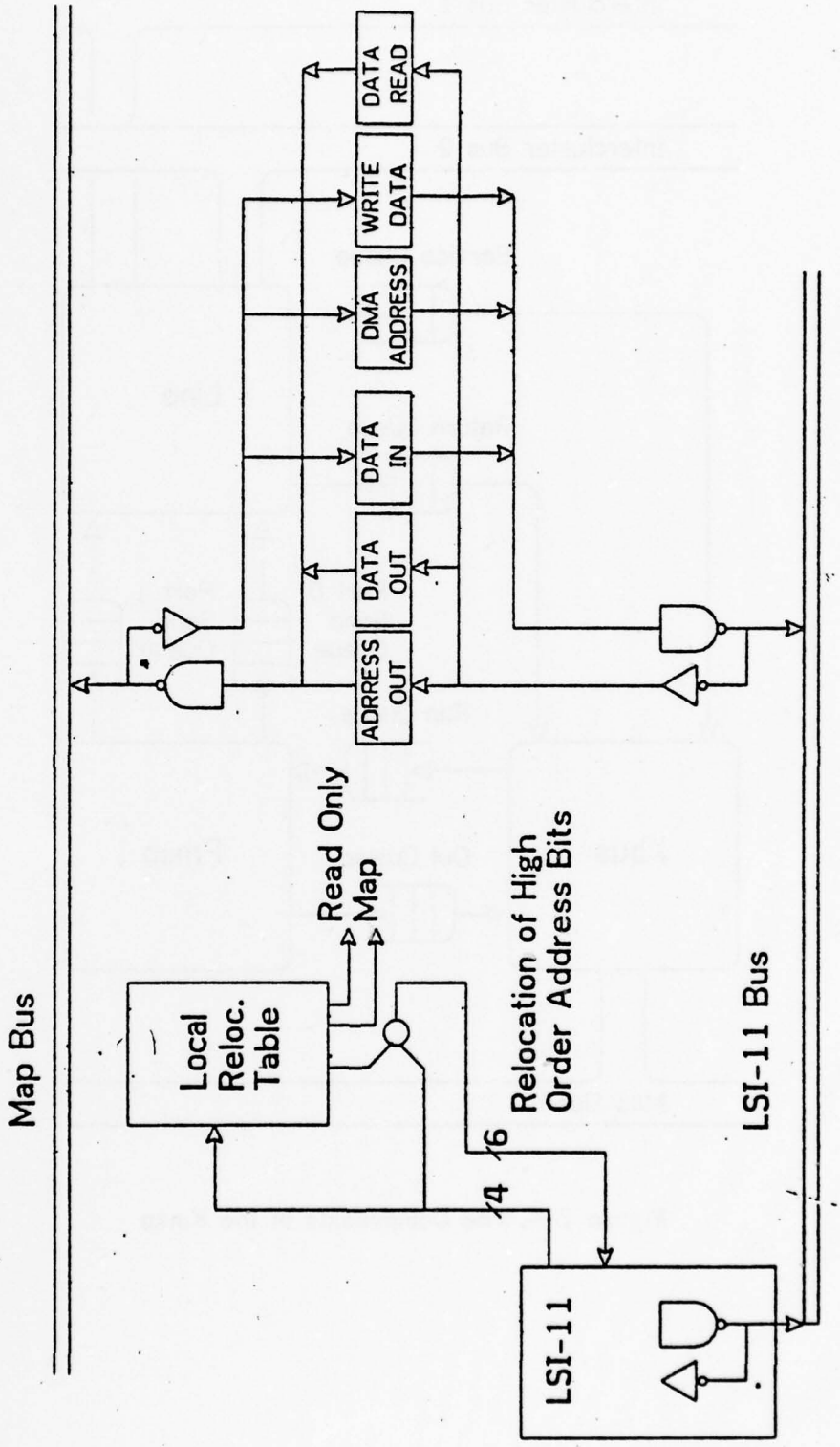


Figure 2-3: A Simplified View of the Slocal

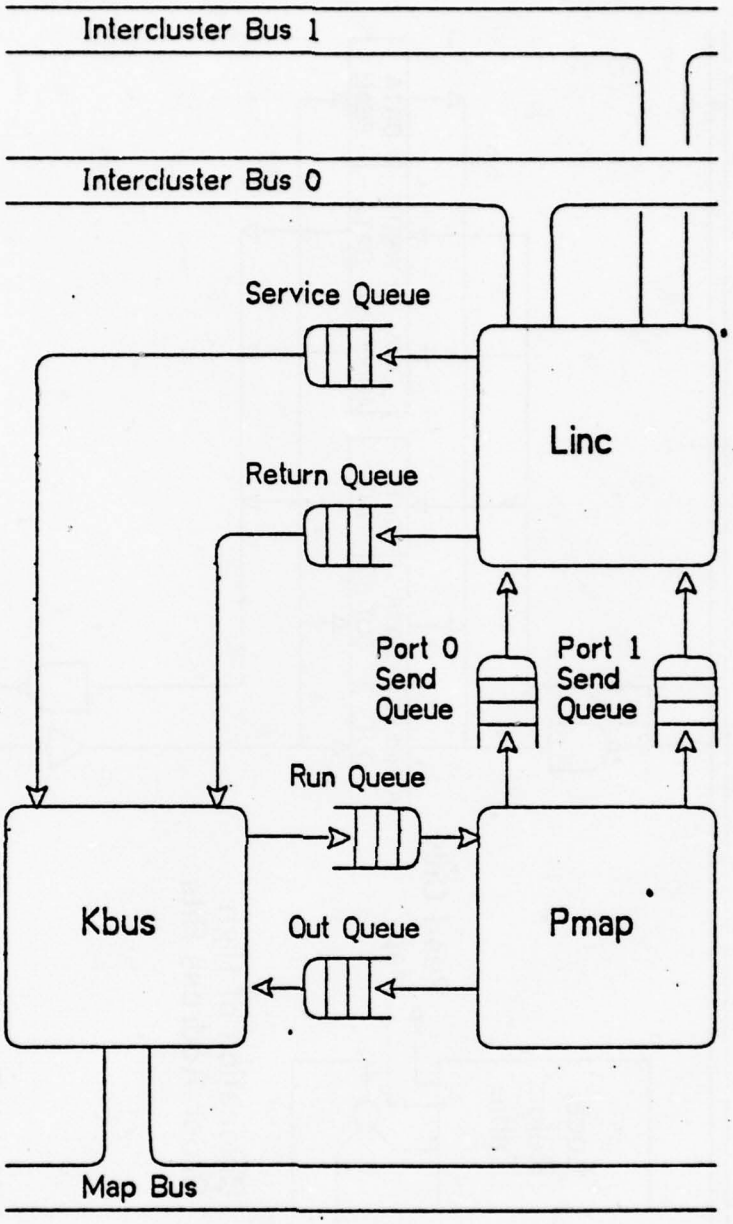


Figure 2-4: The Components of the Kmap

in Section 4.4.)

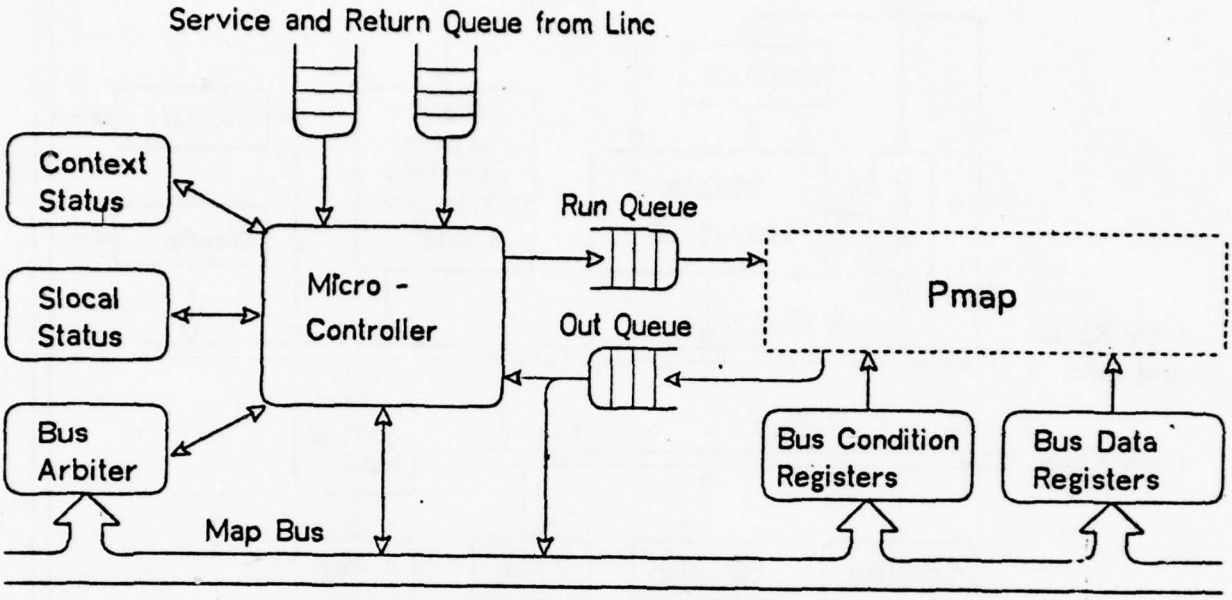


Figure 2-5: The Kbus, or Map Bus Controller

2.4.2 The Multiple Levels of Memory References

Each Computer Module in Cm* is a computer, a processor with memory, in its own right. In normal operation, most memory references (85% to 99%) are by a processor to its local memory. These memory references are performed in a completely standard way. The second possible level of memory reference is by a processor to the local memory of another processor in the cluster. The third level is a reference to the local memory of a processor in another cluster. The fourth, and subsequent levels, occur for references to clusters which can only be reached by forwarding via intermediate clusters. We will consider these situations in turn:

2.4.2.1 Local Memory References

References by a processor to its own local memory are performed in a conventional manner. The only modification made to the standard LSI-11 mechanism is the insertion of a relocation table in the data path for addresses between the processor and the LSI-11 bus.

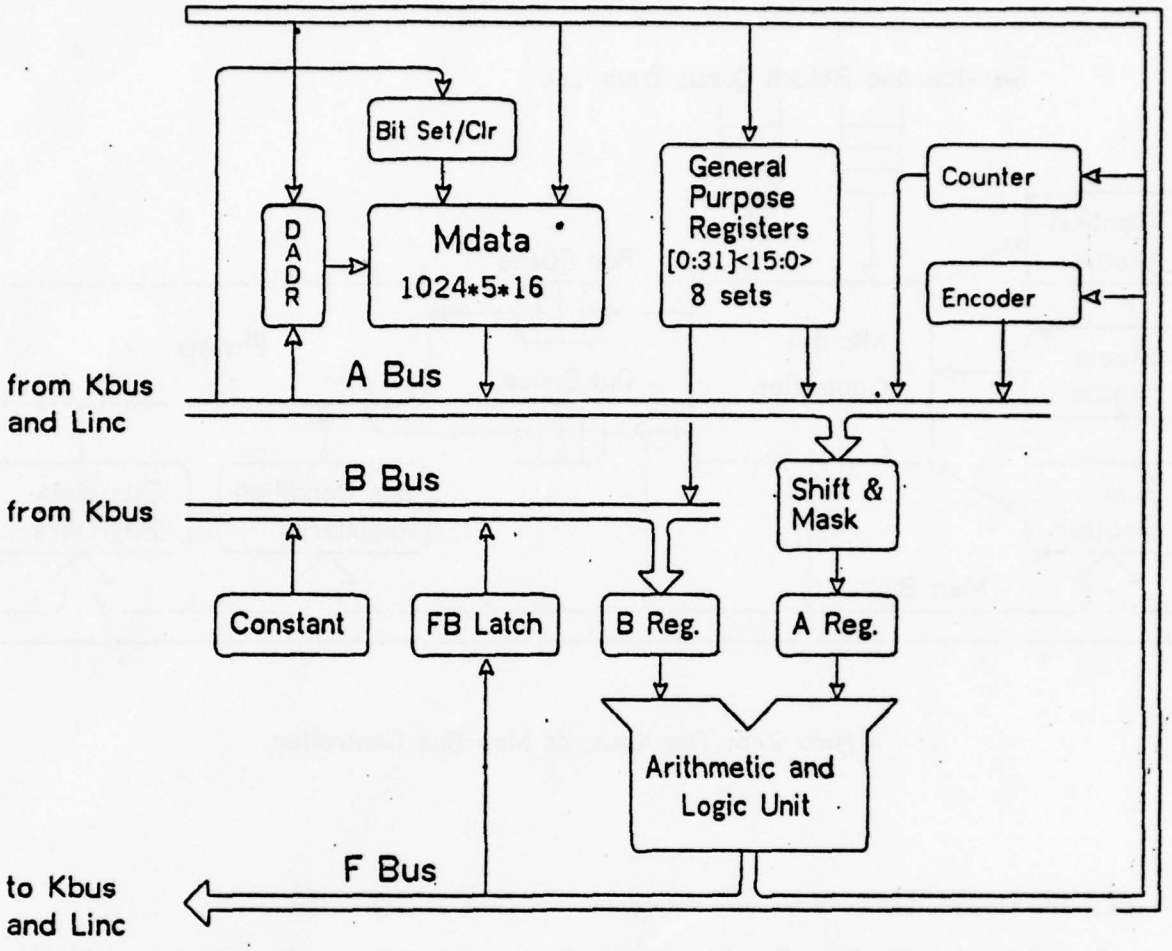


Figure 2-6: Data Paths in the Pmap, or Address Mapping Processor

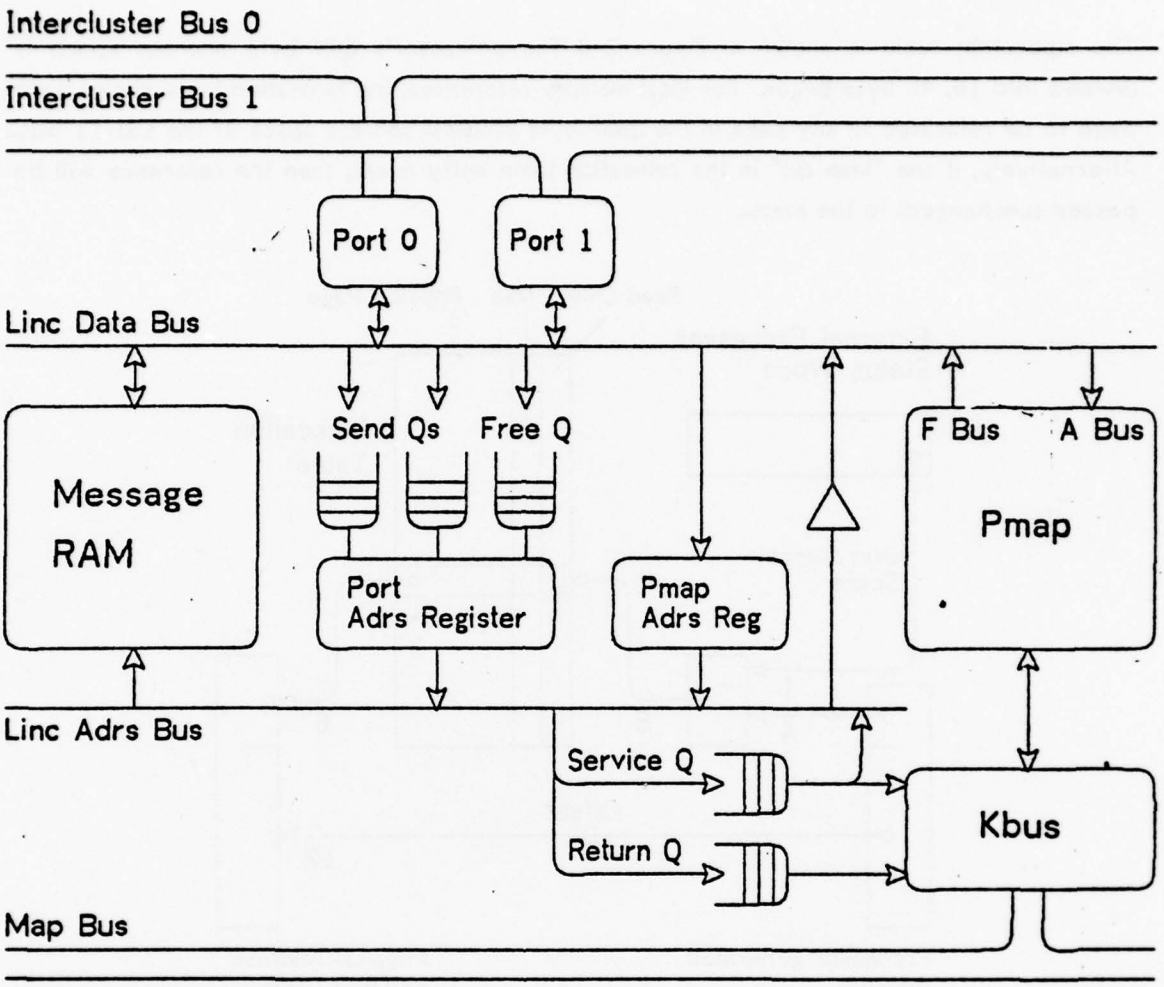


Figure 2-7: The Linc, or Intercluster Bus Interface

The relocation table is shown in Figure 2-8. The processor's 64K byte address space is divided into 16, 4K byte pages. For local memory references, the relocation table allows each page to be relocated to any page in the 256K byte physical address space of the LSI-11 bus. Alternatively, if the "Map Bit" in the relocation table entry is set, then the reference will be passed (unchanged) to the Kmap.

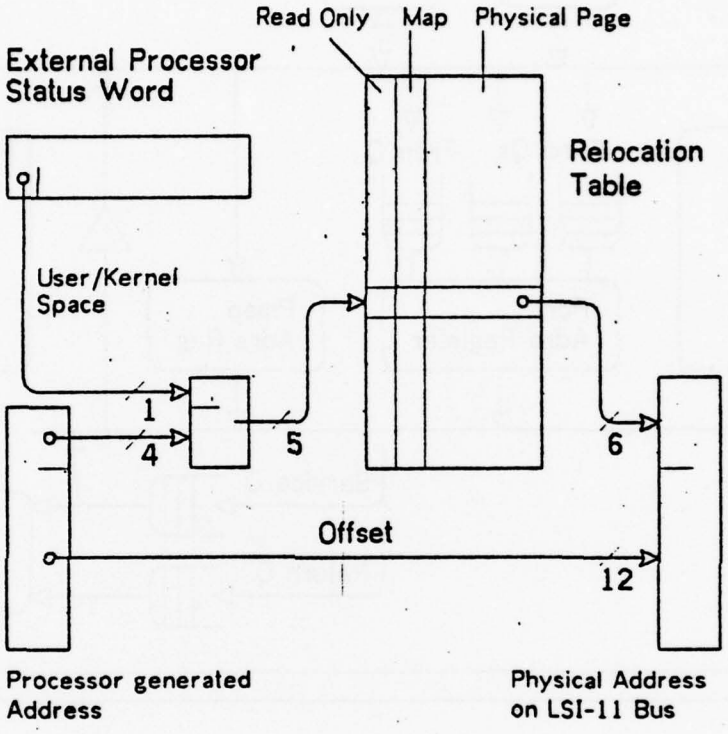


Figure 2-8: Addressing Mechanism for Local Memory References

To give protection for the operating system and interrupt handlers, two independent address spaces are provided for each processor. The current address space is selected by a bit in the eXternal Processor Status Word. The relocation and dual address space system is comparable to the memory mapping facilities available on PDP-11/34s, etc. However, because it was not implemented in the LSI-11 microcomputer it was necessary to provide it with logic on the Slocal. The local relocation mechanism does not slow down the LSI-11 in any way.

2.4.2.2 References Within a Cluster

The relocation table in the Slocal (Figure 2-8) determines which pages in the processor's address space correspond to pages in its local memory and which must be accessed via the Map Bus and Kmap. When the processor references a page which is marked "Map" in the relocation table (see Figure 2-9):

1. The processor generated address is latched by the Slocal and a **Service Request** line to the Kbus is asserted. (Service Request is a line on the Map Bus, see Section 4.3.)

2. For a Write operation the data from the processor is also latched.

3. When a context is available, the Kbus reads the latched address via the Map Bus and places it in a register for that context in the Pmap. The Kbus also places a token for that context in the **Run Queue** to the Pmap. (See Figure 2-5.)

4. When the token reaches the head of the Run Queue, the context is activated. (This is similar to waking up a process in a timesharing system. Context switching takes one microcycle, about 150 ns.) The Pmap takes the 16 bit address from the processor, and on the basis of information in the mapping tables held in the Data RAM, computes the number of the target Cm and an 18 bit physical address within the Cm. (See Chapter 5.) The Pmap places this address and other control information in the **Out Queue** to the Kbus. The Pmap then does the equivalent of a coroutine call to save the current microprogram counter (for possible later reactivation in case of error) and begins execution of the next context (if any) in the Run Queue.

5. The Kbus (which gives precedence to requests from the Pmap) takes the information from the head of the Out Queue. The physical address, with a control bit indicating a read or write operation, is transferred to the Slocal of the designated target Cm. For a write operation, the Kbus performs a second Map Bus transaction, causing the data to be written to be transferred directly from the Slocal of the original requesting Cm to the Slocal of the destination Cm. (The Map Bus is a two address, synchronous bus--see Section 4.3.)

6. The Slocal at the target Cm performs a conventional Direct Memory Access. (The processor is not involved.) When the requested operation is complete, the Slocal signals a **Return Request** to the Kbus. (Return Request is a line on the Map Bus, see Section 4.3.)

If no error condition is indicated, then the Kbus responds to the Return Request by:

1. Causing a direct transfer from the target Slocal to the original requesting Slocal. For read operations this passes the data read back to the requesting processor. For write operations it is simply an acknowledgement that the operation was completed without detected error. In both cases the original source Slocal permits the processor to continue program execution.

2. Marking the corresponding context as free and available for another transaction.

7. If an error condition is detected, the context which invoked the operation is reactivated. This context has retained all information about the request source, virtual address used, physical address generated, etc. Thus the context can respond intelligently to the error situation. For example, the operation may be retried. If the error cannot be overcome, extensive information can be reported back to the originating processor.

2.4.2.3 References to Other Clusters

The principal steps required for a reference to another cluster are shown in Figure 2-11. The Operation of the Linc is shown in more detail in Figure 2-12. Steps A through G in the following description refer to steps marked in Figure 2-12.

1. The request from a source processor for an intercluster reference is indistinguishable from a non-local, intracluster reference. The Kbus allocates a context in the Pmap and the context is activated with the 16 bit address from the processor as a parameter.

2. The Pmap translates the address on the basis of internally held mapping tables, and determines that the reference is to a segment in another cluster.

3. The Pmap prepares a message, containing a complete system-wide virtual address, in the Message RAM of the Linc (Step A). A message also contains the Pmap context number, a 3 bit operation code and a data word (in the case of a write). The maximum size of a message is 8 words, the average size is 2 1/2 16 bit words. (Message formats are shown in Figure 2-10.)
4. Message may be sent over either of the two Intercluster buses connecting to the Linc. The Pmap triggers the transmission of a message by writing the address of the message into the Send queue associated with a particular Intercluster bus. (See Figure 2-7 and Step B in Figure 2-12.) The context handling this transaction is suspended until a response from the target cluster is received (or a timeou' occurs).

5. The Linc in the source cluster is responsible for gaining control of the designated Intercluster bus and transmitting the message word at a time. The Linc in the destination cluster (named in the first word of the message) is responsible for acknowledging the receipt of each word and preparing a copy of

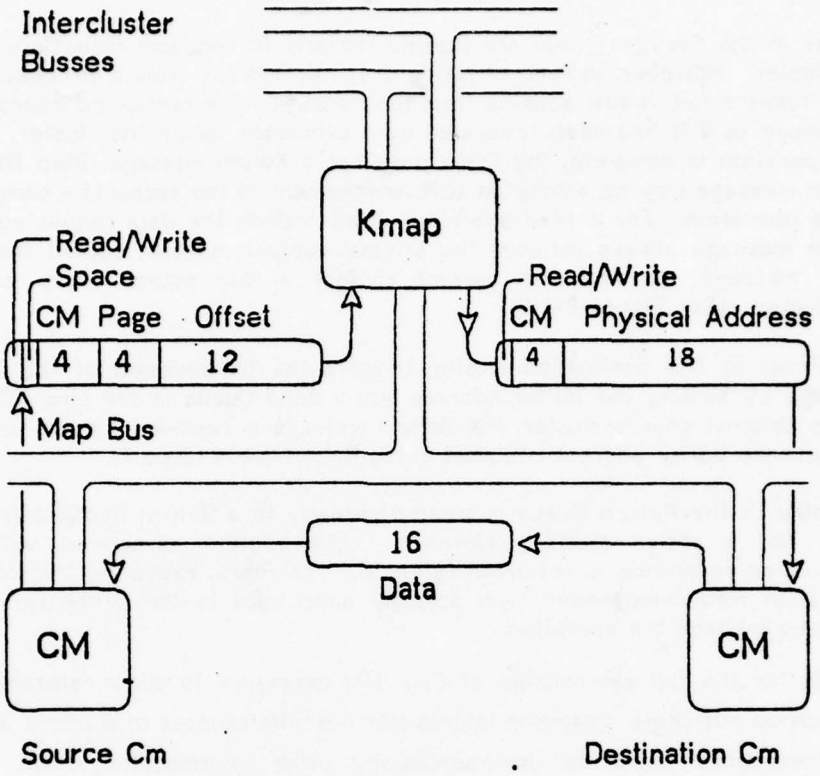


Figure 2-9: A Non-local Reference within a Cluster

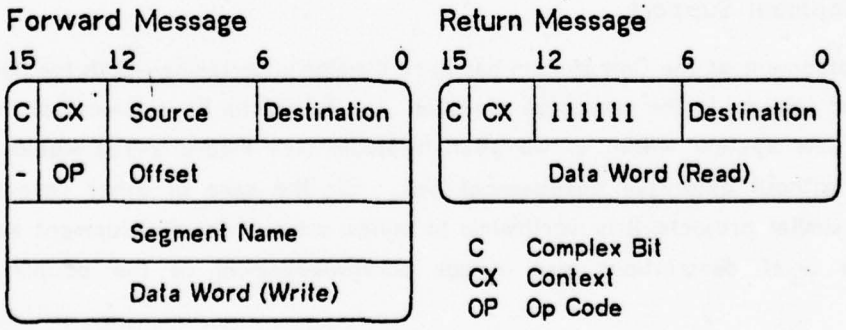


Figure 2-10: Message Formats for Intercluster References

the message in its Message RAM. When the full message is received (Step C), the Linc puts the message buffer address in the Service queue for the Pmap.

6. Entries in the Service Queue are treated similarly to requests from Cm's within the cluster. However, instead of taking a 16 bit address from a processor, the Pmap takes a full virtual address from the message. The requested operation is performed as if it had been requested by a processor within the cluster. When the operation is complete, the Pmap prepares a Return message (Step D). The Return message may be simply an acknowledgement of the successful completion of the operation. For a read operation, it will include the data requested. The Return message always includes the original context number, copied from the initial message, so that the correct context in the source Pmap can be reactivated. (See Figure 2-10.)

7. The Pmap in the destination cluster triggers the transmission of the Return Message by writing the buffer address into a Send Queue in the Linc (Step E). At the original source cluster, the Return message is received by the Linc and the message buffer address is placed in the Return queue (Step F).

8. An entry in the Return Queue is treated similarly to a Return Request from an Slocal for a within cluster reference. The context associated with the intercluster reference is reactived (Step G). The Pmap, executing this context, passes an acknowledgement (and possibly data) back to the processor which originally invoked the operation.

To provide for the full extensibility of Cm* it is necessary to allow references between clusters which do not share a common intercluster bus[1]. References to clusters which are not "immediate neighbors" must be implemented by using intermediate Kmaps to forward messages.

The hardware implementation of intercluster references is explored in more depth in Chapter 3. The issues of address mapping and routing of requests is examined in Chapter 6.

2.5 Development Support

The development of the Cm* system has been a major undertaking--both for hardware and software--in relation to the resources available. The successful development of the initial 10 processor Cm* system within a two year timetable (see Figure 2-13) would have been impossible without extensive development aids. For the sake of other research groups attempting similar projects, it is worthwhile to review some of the development aids used for Cm*; these brief descriptions also include acknowledgement to the people principally concerned.

- The Stanford Drawing Package, which allows logic diagrams to be generated on a graphics terminal and produces wirelists suitable for automatic wirewrap

[1] Without this generality, the total intercluster communication bandwidth would be limited to the bandwidth of one or two individual intercluster buses.

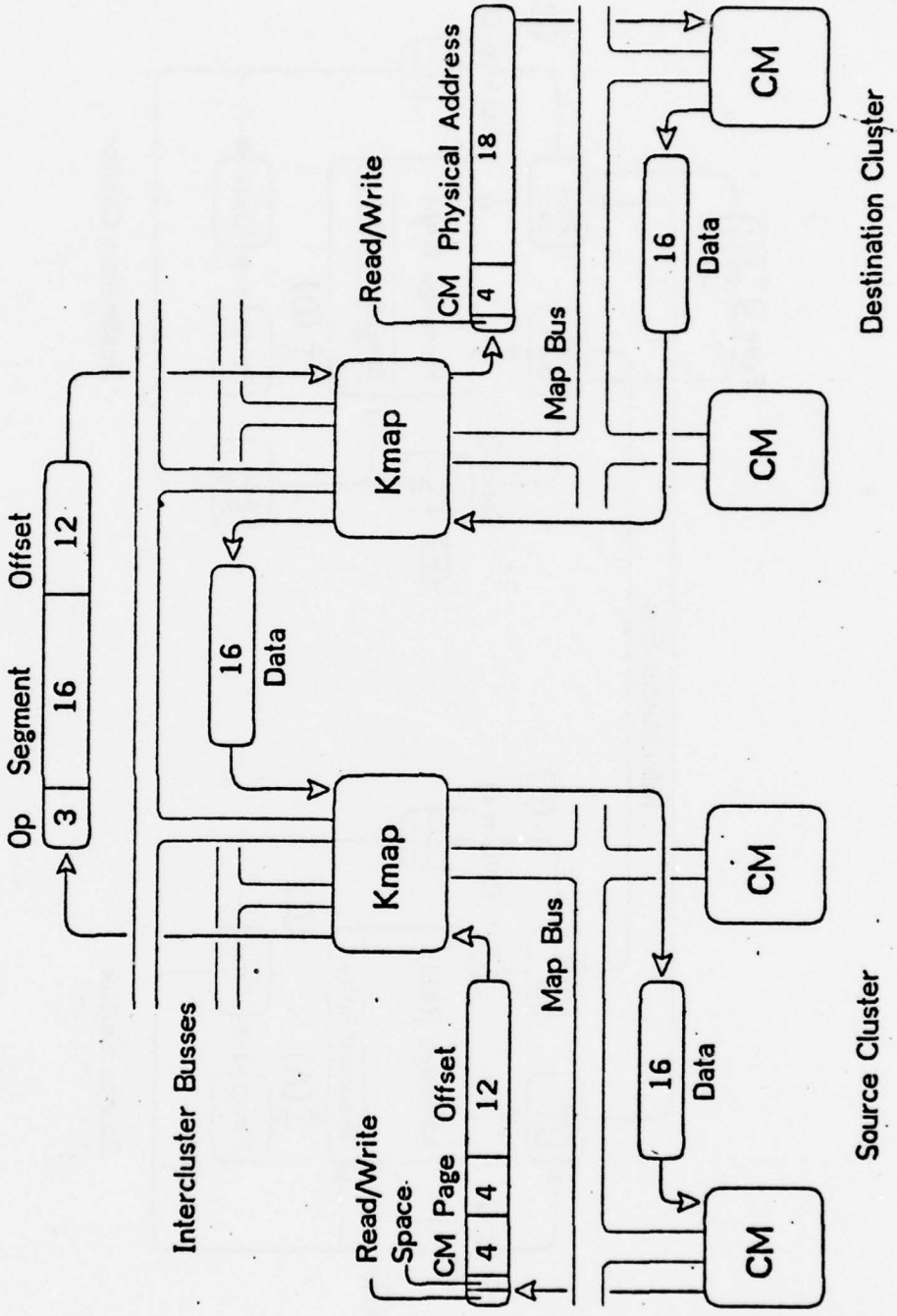


Figure 2-11: An Intercluster Reference

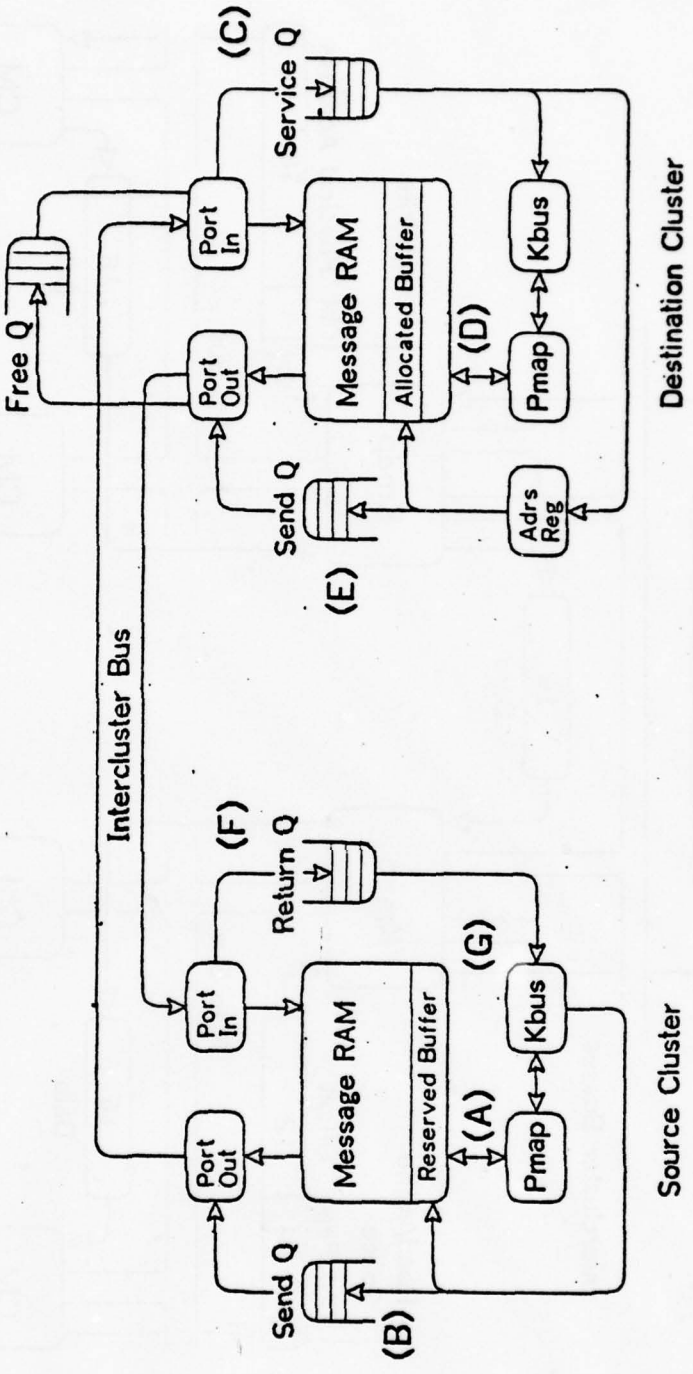


Figure 2-12: An Intercluster Message Transaction

machines, was used for the design of all Cm* hardware components. Relative to conventional manual generation of circuit diagrams, this approach is probably slower for the initial draft. However, it is much faster and vastly more accurate for changing and updating designs. The Drawing Package was originally adapted for CMU by Rick Gumpertz and later maintained by Al Dunlop. A program, SMECO, to merge wirelists and, on the basis of a new and old wirelist, generate a minimum set of wirewrap changes, was written by the author.

- The Pmap is a complex, horizontally microcoded processor. To allow development and experimentation with alternative addressing architectures it has 4K words of writable control store. Typically, microcode is developed with a simulator running on a large computer. (This is the case with the PDP-11/40E's on C.mmp.) This has the advantage of allowing extensive development support. But, a major disadvantage in that it is very difficult, if not impossible, to accurately simulate the environment in which the microcoded processor will operate. To avoid this problem, and to also allow the debugging and diagnosis of the actual hardware, each Kmap is controlled by an independent computer, called the "Hooks" processor. The Hooks processor has full access to the major data paths in the Pmap. It can read and write all registers and the control store. In addition it can stop, start and single step the Pmap and Kbus. Breakpoints can be inserted in the microcode. A support program, KDP , running on the Hooks processor, provides hardware and microcode debugging facilities, a simple microcode assembler, and allows microcode to be loaded directly from a large timesharing system via a serial line. The Hooks processors are at present independent of the LSI-11's within the Cm* switching structure. In the future the "Hooks Interface" may be connected to a processor within Cm*. The additional hardware necessary in the Kmap for the Hooks mechanism is less than 2%. This represents an excellent investment. The Hooks mechanism was conceived, designed and implemented by John Ousterhout.

- Corresponding to the Hooks, which give control of the Kmap, there is a "Host" system which gives control and access to the entire Cm* structure. The Host provides a reasonable user level interface for access to the Cm* hardware, see Figure 2-14. The Cm* Host has a direct serial line and control interface to each processor. With the control interface a processor can be stopped, started and initialized. It is also possible to read the "Run" status of each processor under program control. The serial line connection allows communication with "ODT", a simple debugging system implemented in microcode in a standard LSI-11 processor, loading of programs into the LSI-11 memory, and communication with programs running on Cm*.

  -For hardware development, particularly in the initial stages, the Host system was invaluable because it allowed access to all parts of the structure without the Cm* switching structure being operational. Sitting at a terminal it is possible to access all the state of all computer modules including the internal processor registers, the registers of all devices on the LSI-11 bus (particularly the Slocal) and all memory. The Host system is equally useful for software development. Programs can be loaded either from DEC tapes on the Host or via serial lines from a timesharing system. The Host system was originally developed by Hal Van Zoren and George Robertson. The present version was designed and implemented by Don Scelza and John Ousterhout.

- In a large project like Cm*, requiring the development of a lot of experimental

hardware and software, it is important that programmers have confidence in the correct operation of the hardware. It is difficult enough developing software without the additional burden of discovering design flaws and other hardware faults. Early in the project, an automatic diagnostic system was instituted by Harold Bellis and Dan Siewiorek. In the auto-diagnostic system, one processor attached to the Cm* Host has responsibility for automatically running diagnostics on Cm's which are not in active use. Using the facilities of the Host, this program can assign Cm's, load diagnostic programs, start the diagnostics and log all detected errors etc. With this system, many errors due to IC infant mortality and other causes were detected and reported by the auto-diagnostics rather than causing frustration to programmers. Of course, the effectiveness of the system is limited by the effectiveness of individual diagnostic programs. Another goal of the auto-diagnostic system was to gather statistics on transient failures in digital systems [Siewiorek et al, 78].

- A major problem in the development of any new computer system is to minimize the delay between the availability of hardware and the availability of software to utilize that hardware. The Operating System for Cm*, StarOS [Jones et al, 77], was developed in parallel with the development of hardware and microcode. It was tested by running it on a simulator which ran on C.mmp. By utilizing the writable control stores on C.mmp and the fact that the PDP-11 processors on C.mmp have the same instruction set as the LSI-11's on Cm*, the simulation imposed a performance degradation of only a factor of two. The simulator was implemented by Bob Chansler.

- Apart from program testing and debugging, all software development was done on a large DEC PDP-10 timesharing system. The availability, from the start of the project, of editors, cross compilers and linkers was an obvious, and overwhelming, advantage.

2.6 The Status of Cm

Detailed hardware design of Cm* began in July, 1975. A pilot 10 processor, 3 cluster system was demonstrated with benchmark applications, an Algol 68 language system, and an operating system in June, 1977. The measured performance of five different applications are shown in Figure 2-15. The Integer Programming and Partial Differential Equation tasks showed close to linear speedup as further processors were used. The other applications do not make such effective use of the hardware. The performance degradation is due to the nature of the parallel decompositions and limitations on the size of their input data. Very little of the performance degradation is due to delays in the switching structure. (See Chapter 4.)

Because of an encouraging analysis of the pilot 10 processor system, a 50 processor structure is currently under construction. (A pssobile configuration for Cm*/50 is shown in Figure 2-16.) The larger structure will provide a realistic test of the effective extensibility of the switching structure and test the ability of the architecture and operating system to support close communication and cooperation in a large, distributed multiprocessor.

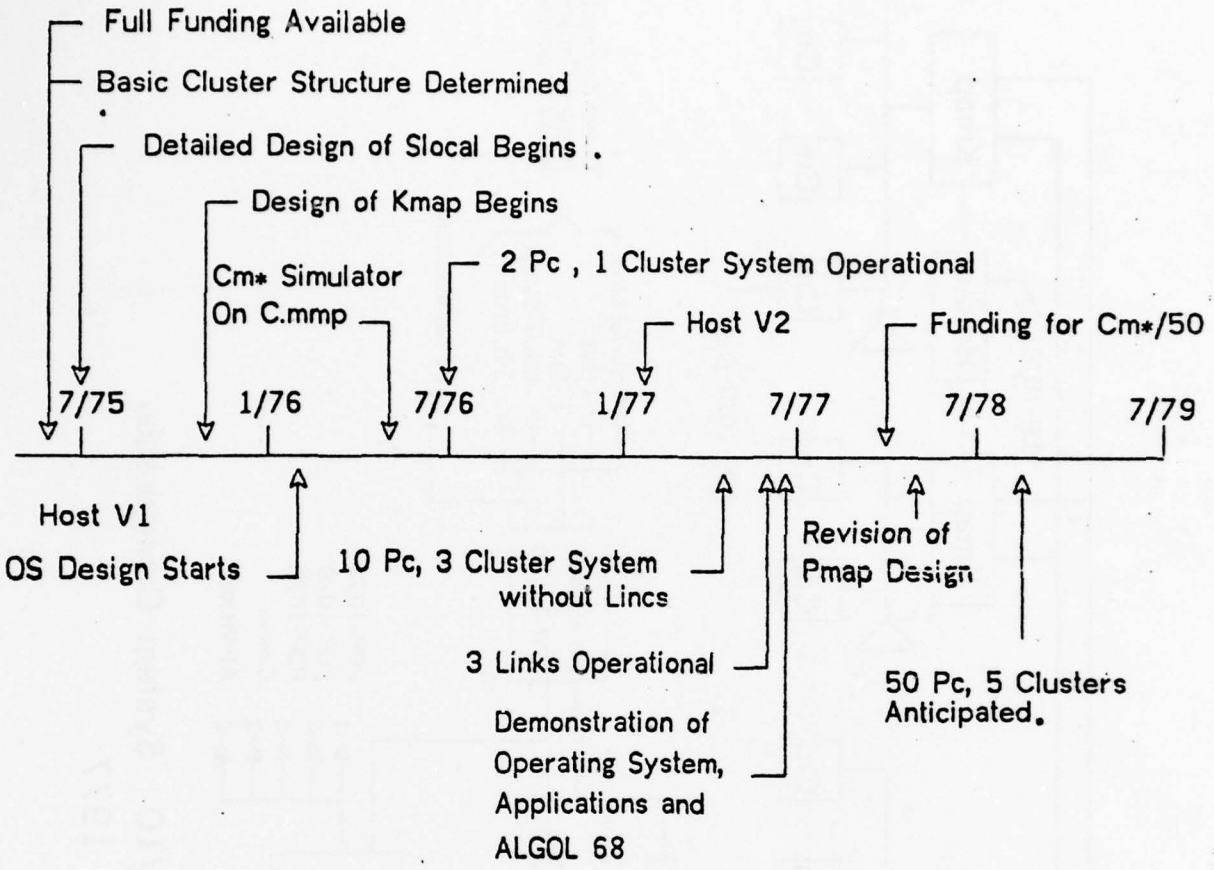


Figure 2-13: The Development of Cm*

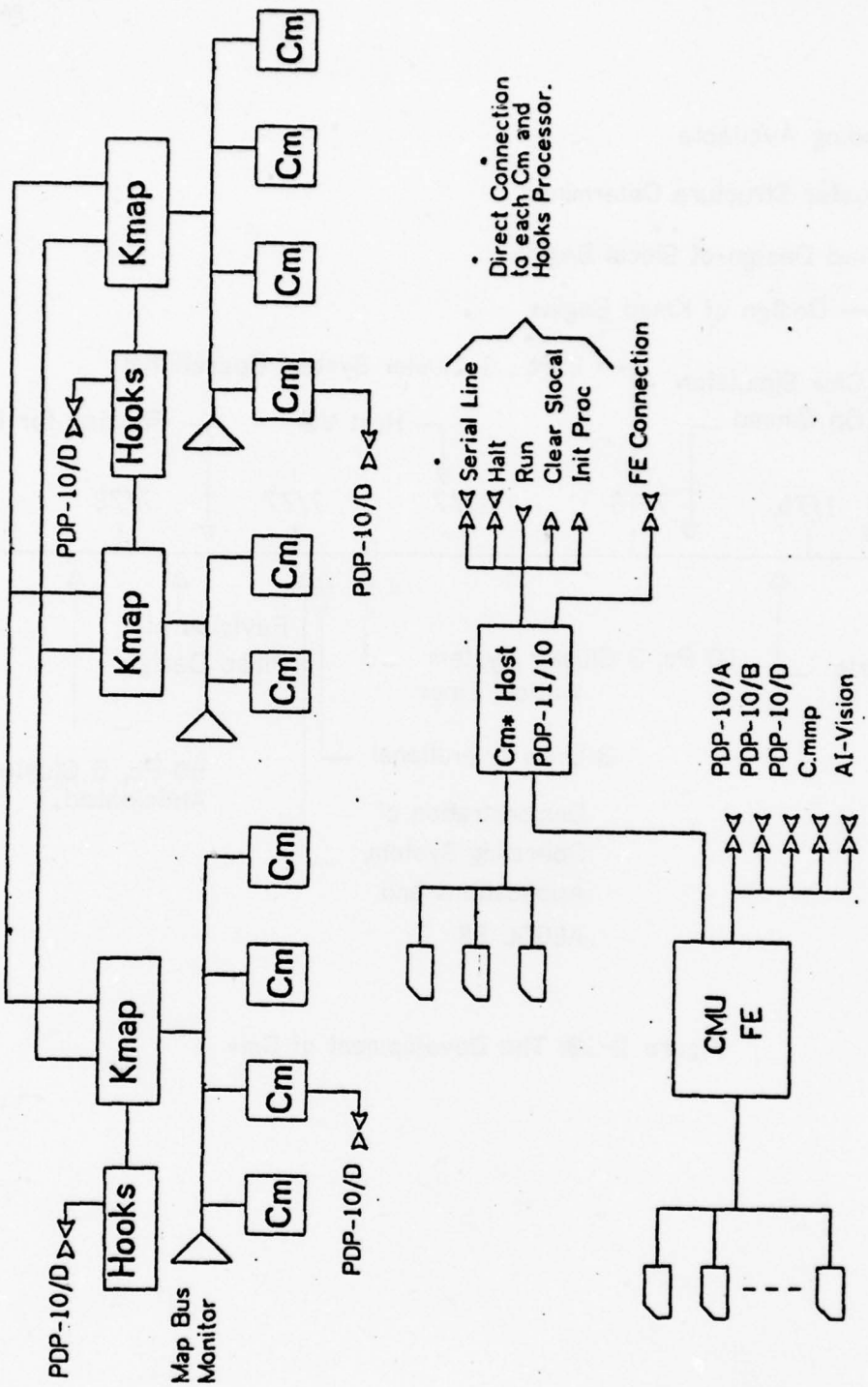


Figure 2-14: The Configuration of Cm*/10

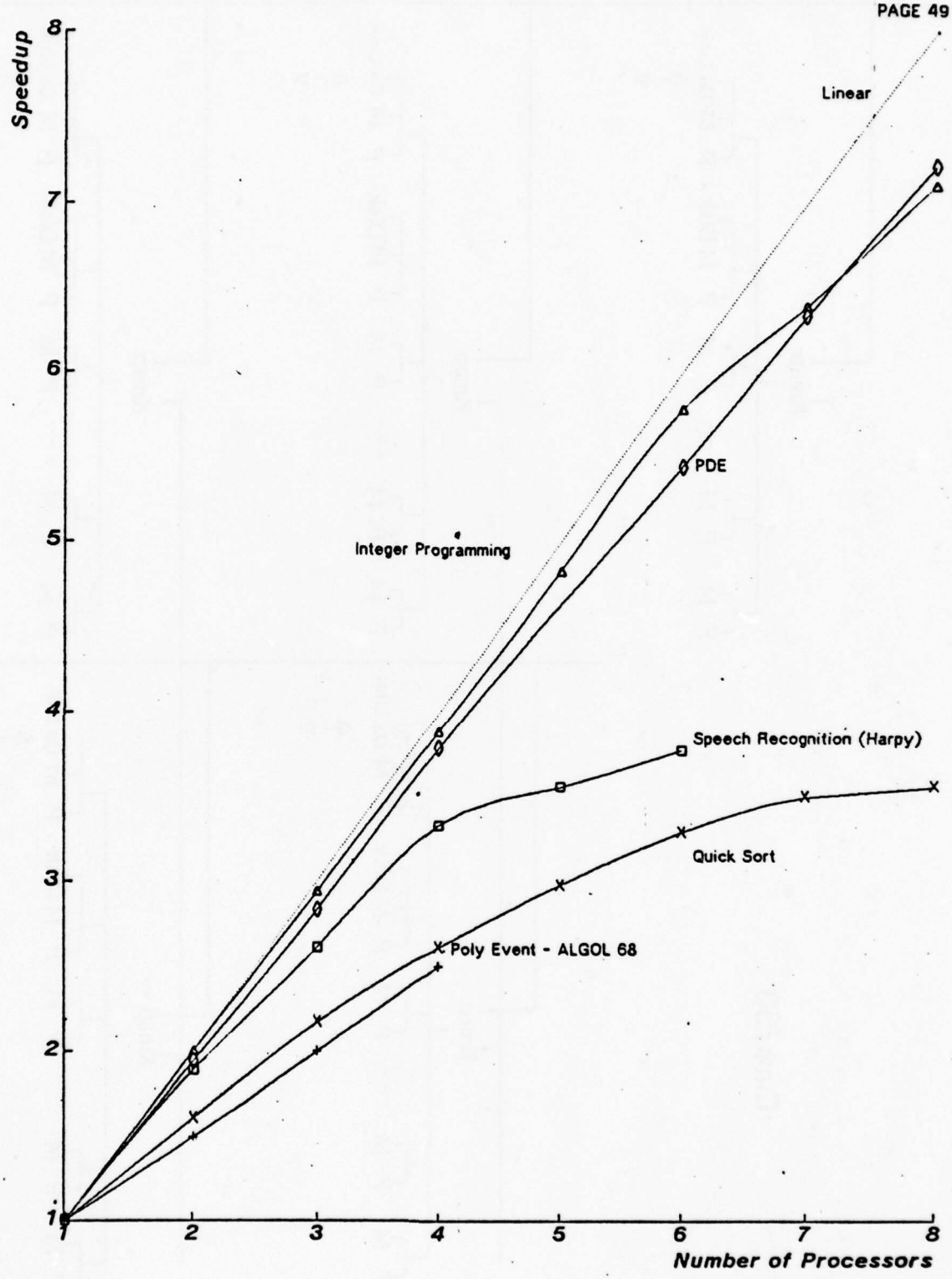


Figure 2-15: The Average Speed-up of Five Algorithms on Cm*

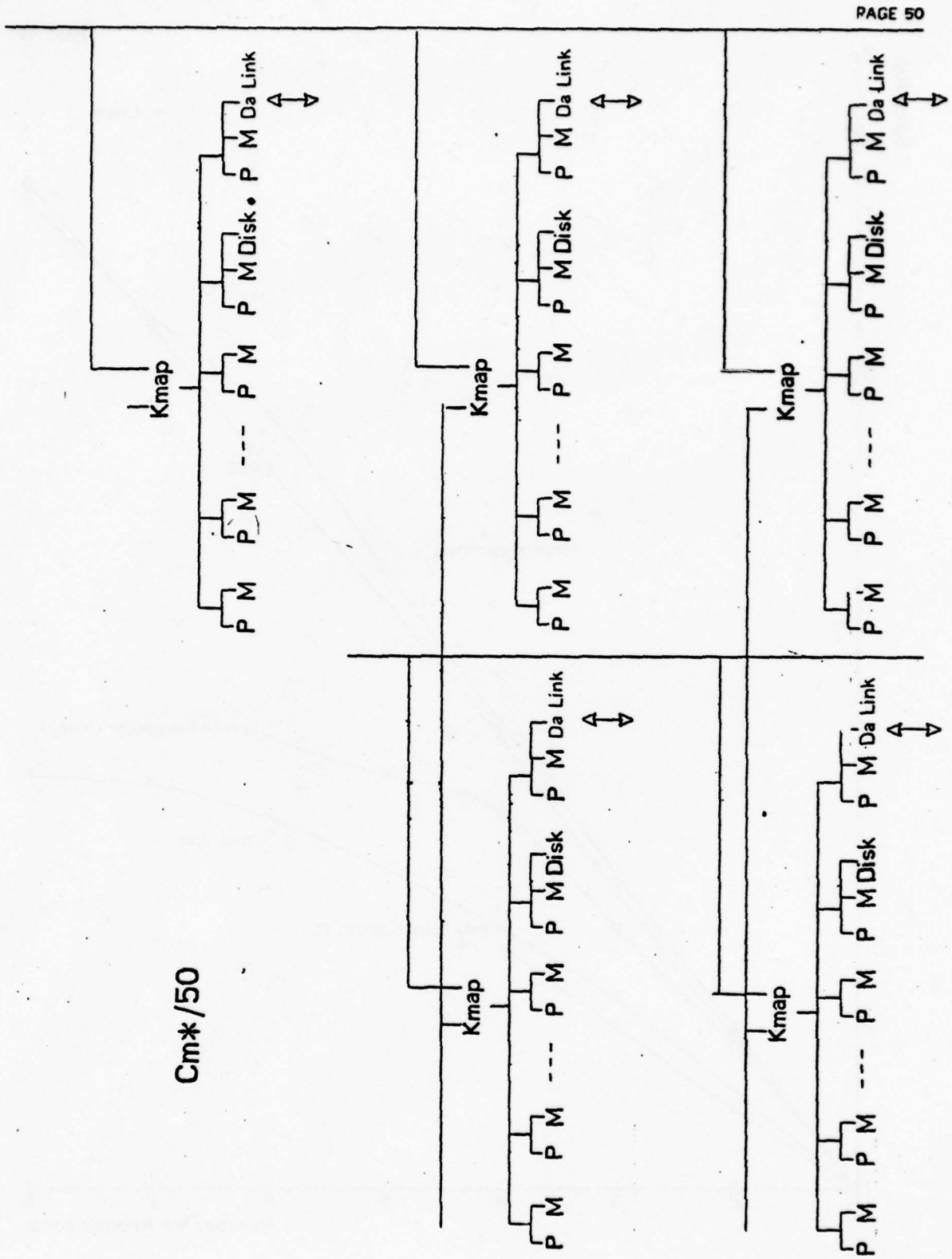


Figure 2-16: A 50 Processor Cm* System

3. Deadlock in Multiprocessor Structures

In the past, multiprocessors have been restricted to simple one level switching structures for access to shared memory. In Chapter 1 it was argued that these conventional structures were ineffective from a cost-performance viewpoint for large numbers of processors. The basic structure of Cm* was introduced in Chapters 1 and 2. An important advantage of the Cm* structure is that its performance can be extended almost indefinitely, with close to linear growth in cost. A disadvantage of the structure is the potential for various forms of deadlock due to interactions between memory references by different processors. This differs from the deadlock problems usually discussed in the literature; deadlocks in a distributed structure must be prevented without a knowledge of the global system state. In addition, resource allocation and deallocation decisions are made at memory reference rates, hence no computational overhead is acceptable. Multiprocessor structures are usually implemented with circuit switching. However, to avoid deadlock in Cm*, this must be replaced with packet switching. The difference between these techniques is described in this chapter. In Chapter 4, it will be shown that packet switching can lead to higher performance, better resource usage, and greater flexibility than circuit switching.

3.1 A Description of Deadlock

In computer science, deadlock is usually discussed with respect to resource allocation in an operating system. At the operating system level, deadlock can occur between two or more processes over the allocation of memory, disk space, magnetic tape units, etc. However, deadlock may occur almost anywhere there is sharing of resources; for example with trains with respect to tracks, with cars in crowded parking lots (particularly in a Pittsburgh winter), with cooking over allocation of burners and stirring spoons, etc. In this chapter we are concerned with deadlock which occurs during a reference to shared memory in a multiprocessor.

The conditions for deadlock [Coffman and Denning, 73] are:

1. **Mutual Exclusion.** Resources are not sharable, i.e., a resource can be allocated to at most one process at any time.

2. **Nonpreemption.** Resources are held until the process finishes using them.

3. **Resource Retention.** A process retains previously allocated resources while waiting to acquire a new resource.

4. **Circular Resource Dependency.** A group of processes is deadlocked if each process is blocked awaiting release of a resource held by another process in the group.

For deadlock to exist, all four conditions must apply simultaneously. Conditions 1 through 3 refer to properties or characteristics of the resources. For example, in modern computer systems, deadlock over memory allocation is prevented[1] by making primary memory preemptible; a process can be halted and swapped out to secondary storage--thus freeing all the primary memory it held. Condition 4, circular resource dependency, is not directly a property of the resources concerned. Circular resource dependency can be prevented by suitable control of resource allocation and limitations on resource requests.

3.1.1 A Simple Example of Deadlock in a Switching Structure

In switching structures, the resources are data paths (or buses) and buffers (or latches), while the processes are individual references by processors to shared memory. The simplest example of this from an actual system is the DEC Unibus Window. Figure (t3a)(a) shows a two processor structure where each processor has part of its address spaced mapped to the local memory of the alternate processor. There are four possible combinations of memory references (1)Pa → Ma and Pb → Mb, (2) Pa → Ma and Pb → Ma, (3) Pa → Mb and Pb → Mb, and (4) Pa →Mb and Pb → Ma. The resources of interest are the two memory buses (Unibuses). If resources are represented by nodes, and the resource needs of a process by directed arcs connecting the nodes, then the four possible combinations can be shown as directed graphs. See Figure 3-1(b). We can see immediately that, during simultaneous references to the memory on the alternate bus, a cycle exists in the Resource Dependency Graph.

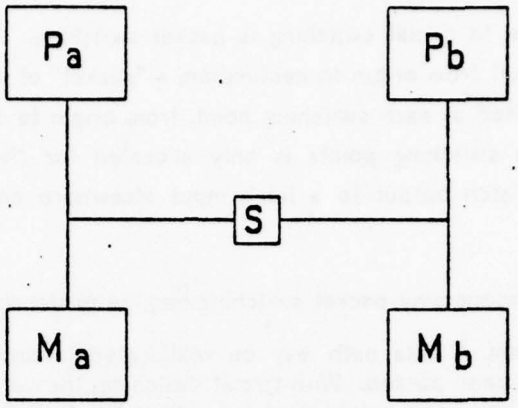
The cycle in the Resource Dependency Graph indicates that deadlock is possible. Intuitively, it shows that if both processors, at approximately the same time, allocate their own memory buses and then request use of the other memory bus (which is already allocated to the other processor) then neither memory reference can complete. Both processors must either wait forever (deadlock) or a time-out must occur and the processors take an error trap (detection and recovery).

3.1.2 Circuit Versus Packet Switching

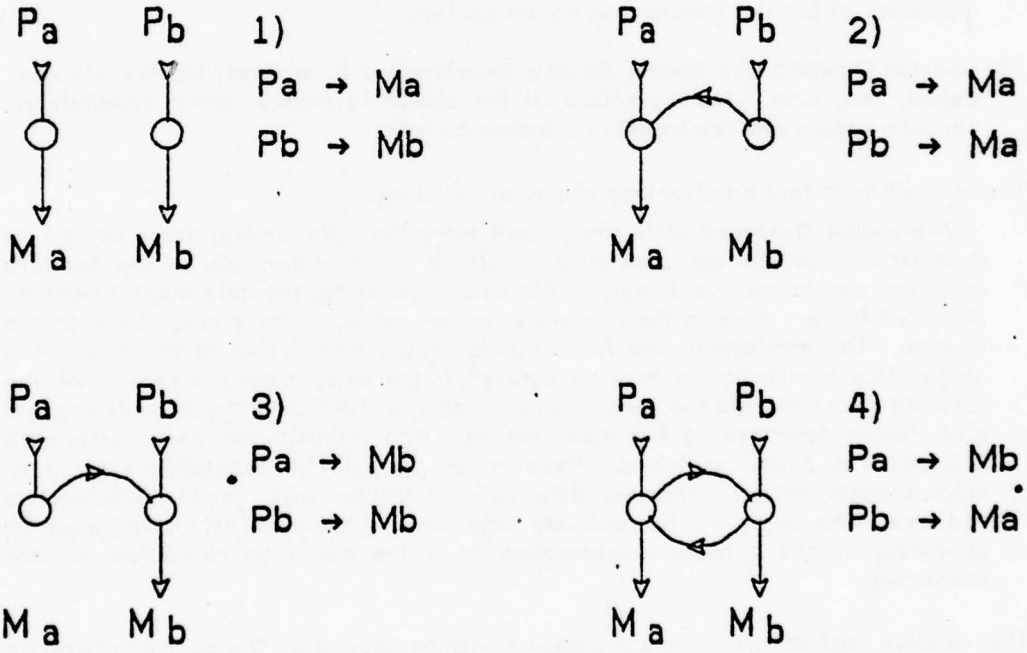
In the example above, and indeed in almost all computer structures, the access path from the processor to memory is circuit switched. That is, a complete electrical circuit is established between the processor and the target memory. Circuit switching is a conceptually very simple and widely used communication technique. It is, for example, used in

---

[1]Some writers distinguish between deadlock *prevention*, where the system design precludes one or more of the deadlock conditions, deadlock *avoidance*, where advance information about resource needs is used to guide resource allocation, and *detection and recovery*, where if deadlock is detected some (usually drastic) recovery procedure is instituted.



(a) Unibus Window Between Two Computers.



(b) Resource Dependency Graphs

Figure 3-1: An Example of Deadlock with the Unibus Window

the public telephone system.

The major alternative to circuit switching is packet switching. Rather than establishing a complete electrical circuit from origin to destination, a "packet" of information is relayed, one step at a time and latched at each switching point, from origin to destination. The electrical path, or bus, between switching points is only allocated for the time period needed to transfer data from a latch output to a latch input elsewhere on the bus (and return an acknowledgement).

There are several reasons why packet switching may be preferable to circuit switching:

1. **Improved Utilization.** A data path may be reallocated to another transaction as soon as data has been passed. With circuit switching, the data path must be held until a full path has been established (i.e., allocating further data paths in the chain) and the desired operation (for example, a memory reference) performed.

2. **Reduced Deadlock Potential.** In a packet switched network, a single transaction never holds more than a single bus. Thus a transaction cannot hold one bus while requesting another and so a cyclic dependency on bus resources cannot develop. (We will see below that deadlock may, however, develop over the allocation of buffers holding information packets.)

3. **Greater Flexibility for Control, Routing, Recovery and Error Reporting.** We will see below, how Cm* takes advantage of the ability to modify, store indefinitely, reroute, return and retransmit information packets.

There is an important disadvantage of packet switching:

Arbitration Overhead. With circuit switching, the data path(s) from source to destination must be allocated once - both to carry information in the forward direction and to receive a reply. With packet switching, the data path(s) must be allocated twice - once in the forward direction and again for a reply back to the source. The arbitration time for a bus is usually comparable to the time for a single data transaction, it may be slower[1]. If the delay from the receipt of the forward information to the generation of a reply is short, then circuit switching will give faster response to individual requests and maintain the same data path utilization as packet switching. However, as the delay in generating a reply or acknowledgement is increased, data path utilization with circuit switching is reduced. This delay will be relatively large when subsequent data paths must be allocated, before a reply is generated, or if the operation requested is time consuming.

The relative cost of circuit versus packet switching depend on the cost of providing data

[1] Arbitration time and data transaction time are closely related because both are limited by the time necessary to reliably deliver a signal to every node on the data path. In many situations, such as DMA arbitration for a conventional memory bus, arbitration is slow because it has been serialized to reduce cost and minimize signal lines.

paths (copper wires, pins on ICs, drive power, etc.) versus the cost of memory and control logic at each switching node. Where memory is expensive (and/or too slow), as in a relay based telephone exchange, circuit switching is the obvious choice because memory usage is minimized. Performance of the alternative schemes depends traffic intensity, arbitration delays, and the time delay in generating a reply after transfer of information in the forward direction. Circuit switching is appropriate for simple structures where the additional control logic and delays inherent in packet switching would dominate. As noted in Chapter 1, present trends show that memory and logic complexity within a module is increasingly cheaper than connections between modules. This technological development will open new areas for the application of packet switching.

3.1.3 A Simple Example of Using Packet Switching to Prevent Deadlock

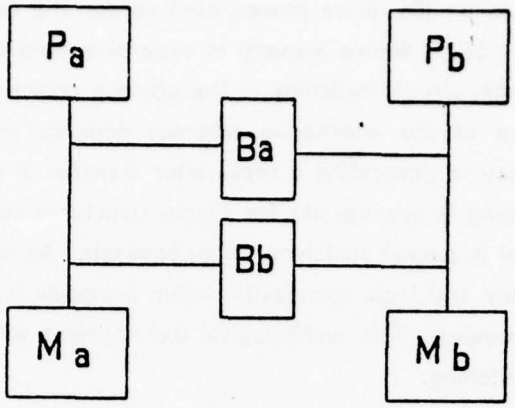
In Section 3.1.1 we showed how deadlock could arise in a simple circuit switched structure. We will now examine a reimplementation of the DEC Unibus Window with packet switching. As noted above, a prime advantage of packet switching is that a bus is never held while requesting a subsequent bus and so buses need not be considered from a deadlock viewpoint. However, the buffer holding a packet at one switching point must be retained while requesting a buffer at the next switching point. Now buffers, rather than buses, are the critical resource for deadlock analysis.

Figure 3-2(a) shows the physical structure of a Unibus window arrangement implemented with latches between the two computers. Figure 3-2(b) shows all possible combinations of memory references. The resources, buffers or latches, are represented as nodes in the resource dependency graphs. There are no cycles in these graphs, hence no deadlock is possible.

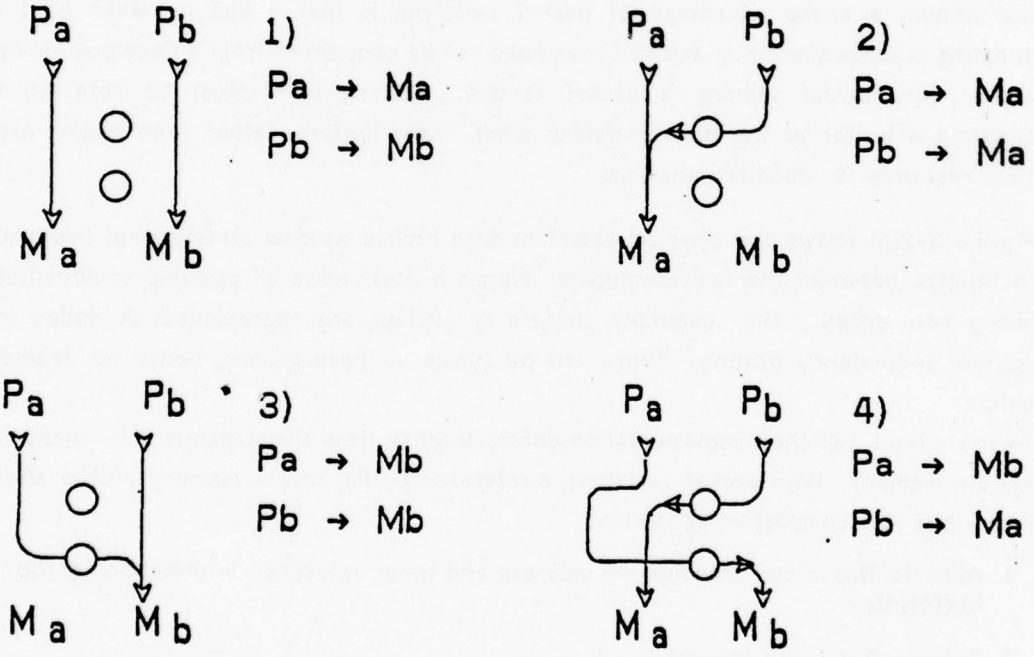
In the circuit switched implementation deadlock arose from simultaneous references to the alternate memory. With packet switching, a reference by Pa, say, to memory on the alternate memory bus is accomplished as follows:

1. Allocate Bus a and transfer the address and other reference information to the buffer, Ba.
2. Release Bus a and request Bus b.
3. When Bus b is allocated, perform the memory reference and latch the result in Ba. Release Bus b.
4. Request Bus a. When allocated, return the result to the processor, Pa.

From an implementation viewpoint, the most marked difference from a conventional



(a) Packet Switched Unibus Window



(b) Resource Dependency Graphs - No Deadlock

Figure 3-2: Unibus Window with Packet Switching

computer structure is that the processor was obliged to release its memory bus before its memory reference, to memory on the other processor's bus, was completed. Conventional processors normally continue to assert address and control information until a response from memory is received. We will see below, the considerable complexity associated with modifying conventional computers to perform this superficially simple operation.

3.1.4 Other Work on Deadlock in Multiprocessors

Most research on deadlock prevention strategies has concerned deadlock in timesharing systems over the allocation of main memory, tape drives, etc. This work is not directly applicable to the Cm* structure because:

1. Cm* is a distributed system with no central arbiter corresponding to an operating system with global knowledge of the status of resource allocation in the system.

2. Resources in Cm* must be allocated and de-allocated at the frequency level of individual memory accesses. Even if global state information was available, the use of traditional allocation algorithms would impose too much time overhead.

However, the very extensive work done on deadlock in operating systems does provide the theoretical background and terminology for the field [Habermann, 69; Holt, 72; Coffman et al, 71]. Bob Chen [Chen, 74] extensively studied deadlock in store-and-forward systems. This has more direct relevance to deadlock problems in Cm* because he considers distributed algorithms which operate without global knowledge. Chen gives some new results for systems with alternate routing of messages. A recent work, [Gavish et al., 77], gives a formulation and heuristics for finding minimal buffer requirements for avoiding store-and-forward deadlock. Previous work has not directly addressed the question of deadlock, at the memory reference level, in an extensible multiprocessor. The work presented here does not include any new theoretical results. However, it includes numerous practical techniques for avoiding deadlock in a distributed system without central control.

3.2 Deadlock Potential in the Cm* Structure

Within the Cm* structure there are three resources over which the issue of deadlock arises:

1. The Local Memory Bus of each Cm.

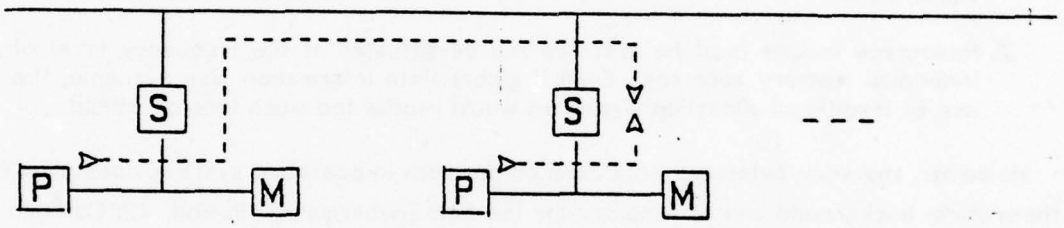
2. Contexts, the multiplexed execution environments in the Pmap.

3. Message Buffers in the Linc used for intercluster references.

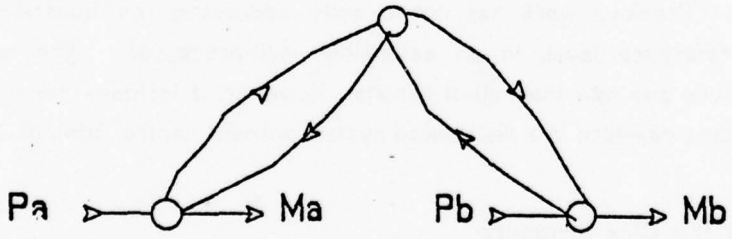
The origin of this deadlock potential, and how it has been eliminated, are discussed in the following sections.

3.3 Deadlock over Memory Bus Allocation

Figure 3-3 shows how deadlock might arise within a single Cm* cluster if two processors attempted, concurrently, to access each others local memory. This is almost identical to the simple example of deadlock given in Section 3.1.1. The solution is also the same; packet switching, rather than circuit switching, is used throughout Cm*. (See Section 3.1.2.)



(a) Deadlock within a Single Cluster Cm* with Circuit Switching



(b) Resource Allocation for Buses within a Cluster

Figure 3-3: Deadlock with Circuit Switching in a Cm*-like structure

Figure 3-4 shows the registers used to buffer the address, data and control signals at each Slocal (the Map Bus-LSI-11 Bus interface). Note that for each Cm there are two sets of buffer registers. One set is for a transaction generated by the local processor for a non-

local memory reference. The other set is for a transaction originating at another processor. (Incoming references are performed by Direct Memory Access, DMA). Two register sets are necessary to avoid deadlock over register allocation during mutual concurrent references.

A crucial factor in correctly implementing packet switching at the memory reference level is that the processor not hold its local memory bus during remote references. The problems associated with this are discussed in the following sections.

3.3.1 Packet Switching within a Cm

Figure 3-5(a) shows a possible internal structure for a Cm which implements packet switching. There is a structural separation of devices which generates memory references (active devices, or sources--for example, processors and the Direct Memory Access portion of a disk controller) from passive devices which respond to references, for example memory and device control registers. We note the following properties of this structure:

1. References on the Active Device Bus must contain at least enough information to select between being latched for transfer over the Intracluster Bus or making a direct reference to the Passive Device Bus.

2. References (necessarily from the Active Device Bus) to non-local memory do not involve the local memory bus. Once a reference (from the local processor or elsewhere) has been allocated the local memory bus it requires no further resources and so is guaranteed to complete within a short time. Thus there can be no deadlock over allocation of the local memory bus.

3. The Active Device Bus is used only by the devices on the bus--it is not accessible from anywhere else in the system. Also, it is the first step in any reference path. Hence any devices seeking it cannot be holding any other resources and so the Active Device Bus cannot be part of any cyclic resource dependency chain causing deadlock.

4. Similar, simple arguments will show that the other shared resources in the Cm, the buffers in the interfaces to the Intracluster Bus, cannot be involved in any deadlock chain. Thus the structure allows sharing of local memory without deadlock--at least within a single cluster.

5. It is easy to see that a device such as a disk controller acts both as passive device, for communication of commands and status, and as an active device for direct transfers between secondary storage and any memory in the system. The processor, however, may mistakenly be thought of as a purely active device. In a multiprocessor structure it is necessary for control operations to be performed by one processor on another. Examples include interprocessor interrupts, halt, reset, single step, etc. Control operations which would normally be part of circuit switched bus protocol may also be implemented via this general control mechanism, for example, the reporting of parity, non existent memory, and invalid operation errors detected during a memory reference.

6. References to local memory are not packet switched, they are performed in a

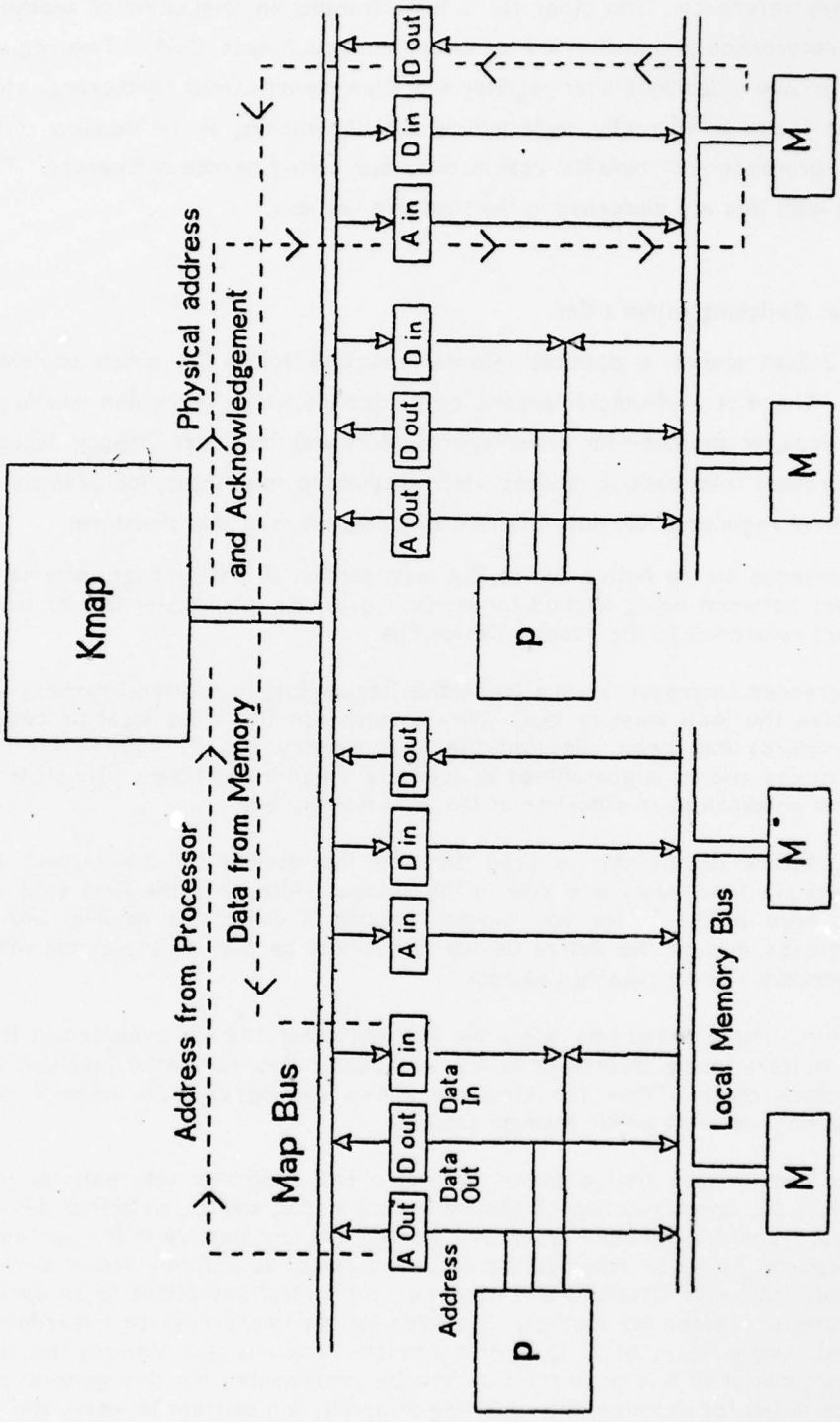
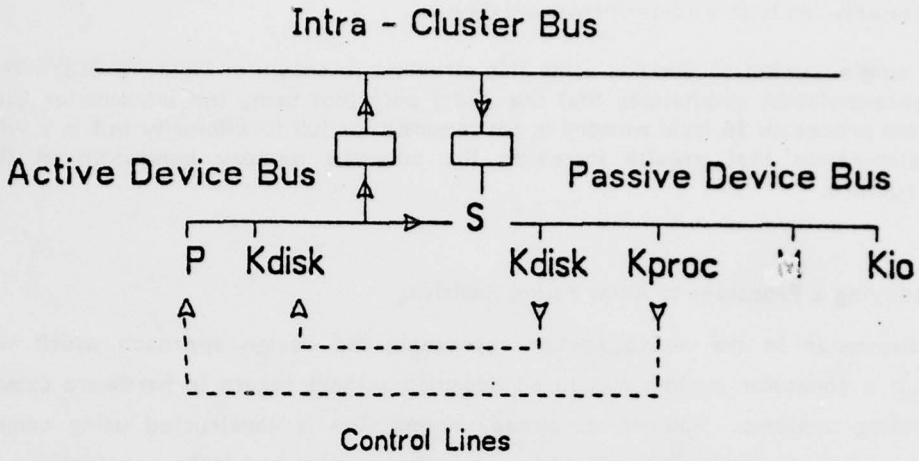
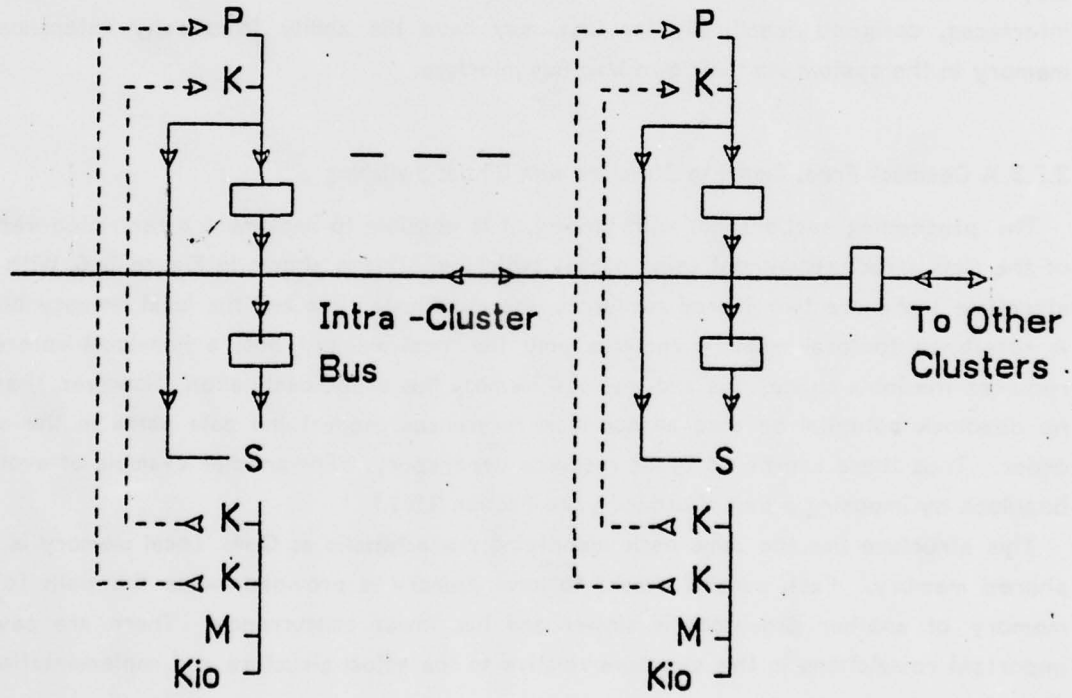


Figure 3-4: A Simplified Representation of Data Paths in a Cluster



(a) Ideal Cm Structure for Packet Switching



(b) Single Cluster Built from Ideal Cms

Figure 3-5: Ideal, Packet-Switched Structure for Cm*

conventional, circuit switched way. This presents no deadlock problem and probably leads to a faster implementation.

7. A single cluster of modules using this structure is shown in Figure 3-5(b). This representation emphasizes that the direct path (not using the Intracluster Bus) from processor to local memory is not required for full functionality but is a vital optimization that greatly increases the potential memory bandwidth of the structure.

3.3.2 Modifying a Processor to Allow Packet Switching

The discussion in the above section represents the design approach which would be adopted if a computer system was to be designed without regard to hardware compatibility with existing systems. Howver, as already noted, Cm* is constructed using commercially available microprocessors (DEC LSI-11s). These processors had to be substantially modified to prevent the processor from holding the bus during a non-local memory reference. These modifications are described in Section 4.6.2. Because of the wide range of devices involved, no DMA devices have been modified in this way. Thus conventional disk interfaces etc. can only do direct transfers to and from memory on the local memory bus. Future device interfaces, designed specifically for Cm*, may have the ability to directly reference all memory in the system via their own Map Bus interface.

3.3.3 A Deadlock Free, Cm*-like Structure with Circuit Switching

The preceeding sections not withstanding, it is possible to implement a restricted version of the Cm* structure without using packet switching. This is shown in Figure 3-6. With this structure there are two shared resources, the intracluster bus and the local memory buses. A reference to local memory requires only the local memory bus; a non-local reference requires the intra cluster bus and the local memory bus at the destination. However, there is no deadlock potential because all non-local references request the data paths in the same order. Thus there can be no cyclic resource dependency. (For another example of avoiding deadlock by imposing a partial ordering see Section 3.5.1.)

This structure has the same basic identifying characteristic as Cm*: Local memory is also shared memory. Fast, parallel access to local memory is provided, while the path to the memory of another processor is slower and has lower concurrency. There are several important restrictions in this structure relative to the actual structure and implementation of Cm*.

1. The structure is not extensible to multiclusters. A multicluster structure would deadlock over the allocation of intra cluster buses. With multiclusters a partial ordering cannot be imposed on bus allocation. (A possible approach to resolving this deadlock would be to restrict the interconnections to a tree structure rather

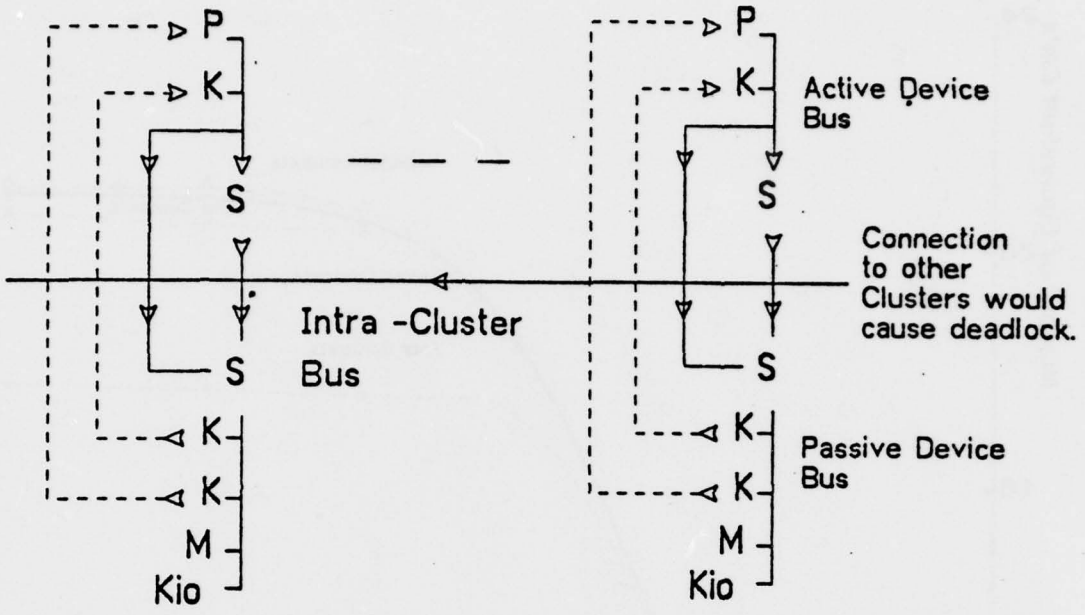


Figure 3-6: A Circuit-Switched, Deadlock Free, Single Cluster Cm* like System

than the arbitrary graph structure of Cm*. This would allow direct detection of deadlock and some form of abort and retry might be used.)

2. Concurrency of non-local references is exactly one. Whereas the packet switching of Cm* gives a concurrency of about five, using the same datapath width and technology. The curve for the number of contexts being equal to 1 in Figure 3-7 shows the effect a concurrency of 1 would have on the performance of Cm*. This substantial degradation does not necessarily apply to a different implementation where the memory bus arbitration at the target memory was substantially faster.
3. The powerful address mapping and routing mechanisms of Cm* would be difficult, if not impossible, to implement in a circuit switched structure.

Despite these cited limitations, the structure is simple, cheap to implement and well suited to many dedicated control tasks. It provides a strong alternative to Pluribus and other full crosspoint multiprocessors with respect to cost while giving far greater flexibility than closely coupled, specialized networks often used for control applications. Special devices requiring a high memory bandwidth, such as graphic displays, can be interfaced to multiple memory buses. A version of this structure, using Texas Instrument 9900 microprocessors, has been designed in consultation with the author. It will be used by a commercial firm for

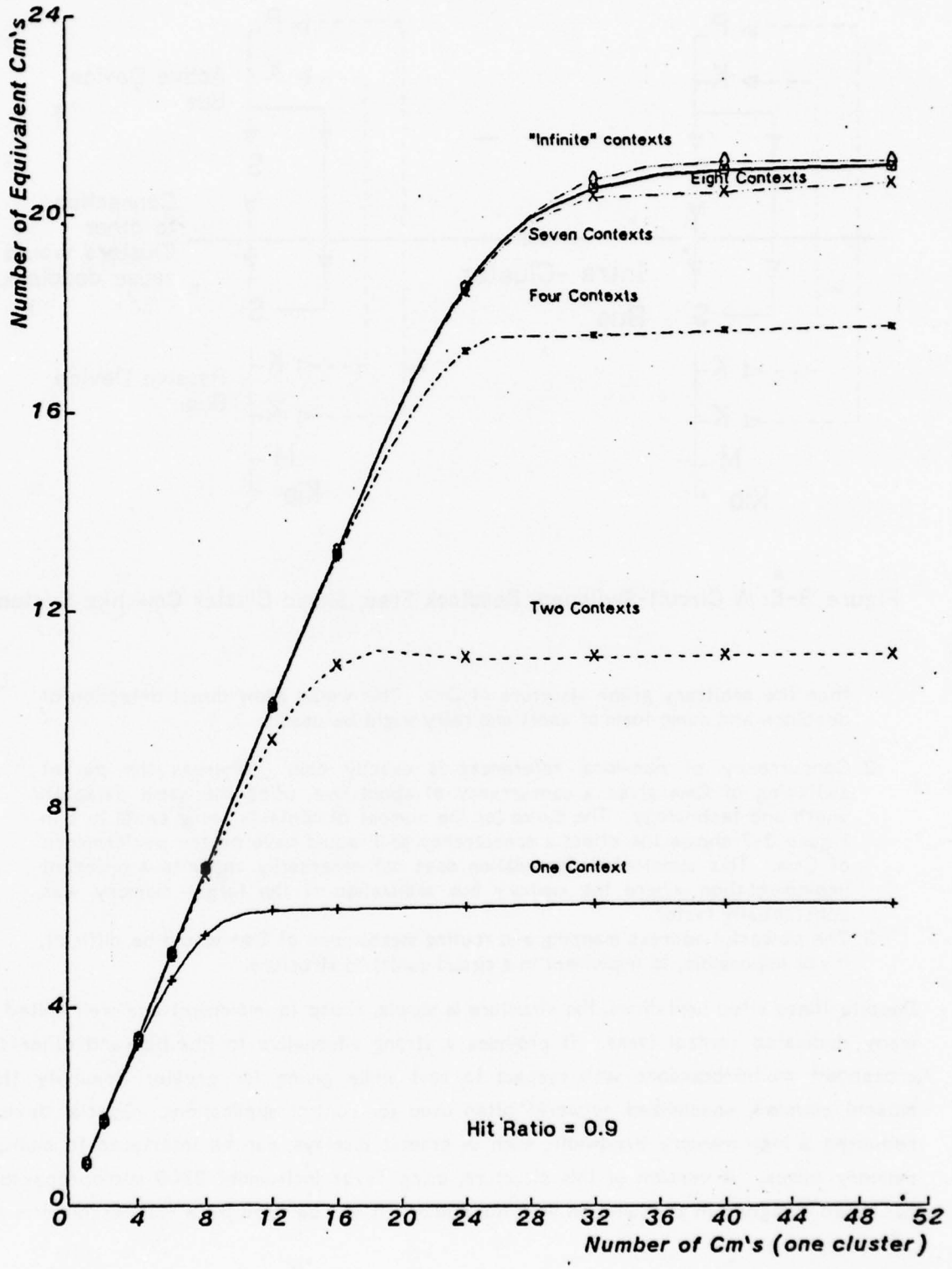


Figure 3-7: Effect of Number of Contexts on Performance (Simulation)

communications and real-time control applications.

3.3.4 Generalized, Partial Ordering on Bus Allocation

The previous section showed how a single cluster Cm*-like system could be implemented with circuit switching by enforcing a partial ordering on bus allocation. This mechanism can be generalized to systems with more than two levels of buses. Figure 3-8 shows a multicluster Cm*-like system with four levels of buses. Level 0 is the local memory bus, level 1 is the intracluster bus, level 2 is an intercluster bus, level 3 connects groups of clusters. Deadlock over bus allocation can be prevented in this structure by ensuring that buses are allocated in descending numerical order. (This defines a partial ordering on resource allocation, thus no cyclic resource dependencies can arise.) With this approach, for a processor in one cluster to access memory in another adjacent cluster, the order of bus allocation is: First the intercluster bus, then the two intracluster buses (in any order), and then the two local memory buses (again order is unimportant).

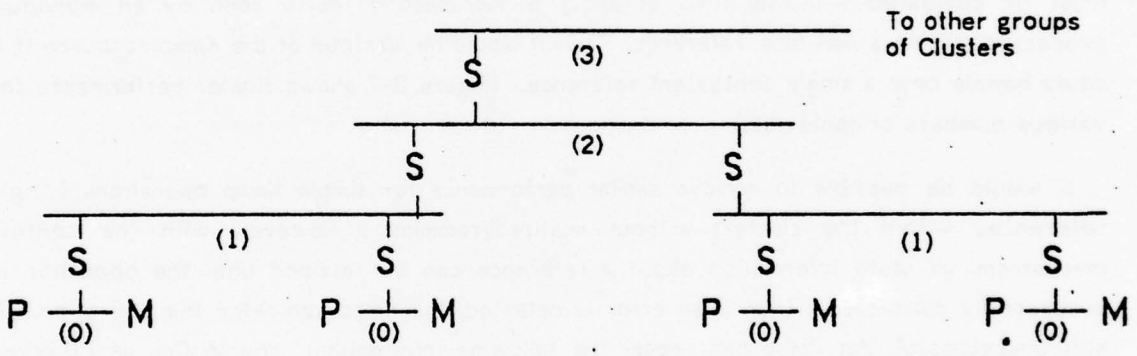


Figure 3-8: Deadlock Free, Many Leveled, Circuit Switched Structure

This system, with many levels in the bus hierarchy, is probably not practicable. The bus allocation mechanisms would be quite complex. However, this discussion does show that the structure shown in Figure 3-6 can be simplified. The active device bus - passive device bus separation is not necessary. There can be a single, conventional local memory bus with processor and DMA devices for each "computer module". To avoid deadlock on non-local references, the intracluster bus must be allocated prior to the allocation of the requesting processor's local memory bus. This can be easly implemented.

3.4 Deadlock over Pmap-Context Allocation

As mentioned in Chapter 2, the Pmap is 8-way multiprogrammed (in hardware) Each independent set of registers and program counter is called a context. Contexts are allocated both for non-local references by Cm's within the cluster and for references from remote clusters received via the Linc. The context mechanism is a central part of the Kmap design. (A similar mechanism is used in the Alto [REF ****], a computer designed at the Xerox Palo Alto Research Center. In the Alto, the hardware multiprogramming is used to drive a raster-scan display and to control other I/O devices.)

3.4.1 Motivations for the Context Mechanism

The motivations for the context mechanism in the Kmap, rather than a simpler and more conventional design, are a combination of performance, error control and ease of microcoding for complex (multiple memory reference) operations. It takes 6 to 10 microinstructions (1-1.5 microseconds) in the Kmap to do the address mapping for a simple memory reference. This must be compared with the total of about 6 microseconds delay seen by an individual processor making a non-local reference. Thus it would be wasteful of the Kmap resource if it could handle only a single concurrent reference. (Figure 3-7 shows cluster performance for various numbers of contexts[1].)

It would be possible to achieve similar performance for simple Kmap operations (single references within the cluster) without multiprogramming. However, with the context mechanism, all state information about a reference can be retained until the operation is successfully completed. Thus if an error is detected, the Kmap can retry the operation. If still unsuccessful, the Pmap can report the following information: source Cm, user/kernel, source "virtual address", physical address generated, destination Cm, nature of error--e.g., protection violation, memory parity, non existent memory, Map Bus parity, non existent Cm. The flexibility inherent in handling the error response in the Kmap allows programs to dynamically specify for each error class the mechanism used to report the error. For example, a user program may wish to handle parity errors by a trap to a user routine while protection violations and non existent memory problems cause a trap to the O.S. An initialization routine within the operating system may wish to detect non existent memory by explicitly testing a flag rather than taking an error trap.

The second compelling reason for the hardware supported multiprogramming is to support Kmap operations which require multiple memory references or references to other clusters.

---

[1] This graph was generated by Levy Raskin, and is based on an accurately calibrated simulation model.

Even comparatively simple operations, such as an indivisible decrement operation, require two memory references. In the present microcode, making a segment addressable may generate 22 memory references. There would be an intolerable loss of performance if concurrent requests to the Kmap were blocked for these periods. To make these operations feasible it is necessary to multiprogram the Kmap. Multiprogramming could be achieved in microcode--with a considerable loss of performance and increase in microcode complexity. The cost of hardware to support the context mechanism is less than 5% of the Kmap and appears to have been a sound investment.

3.4.2 Choice of the Number of Contexts

The choice of the number of contexts is mostly a question of the useful level of concurrency possible in the Kmap and Map Bus combination. Figure 3-7 shows that a single cluster can support on the order of 5 concurrent within-cluster references. Beyond this, the Map Bus saturates. Thus for simple operations, 5 contexts should be sufficient. More complex operations, and particularly references outside the cluster, may use additional contexts.

3.4.3 Context Allocation for Intra Cluster References

There is no question of deadlock over context allocation for references within a cluster. There cannot be deadlock because no process (memory reference operation) requires more than one context to complete. When all contexts are in use, subsequent requests to the Kmap are simply blocked. The Kbus, which performs all Map Bus and Pmap arbitration, implements a pseudo round-robin allocation policy to prevent starvation.

3.4.4 Context Allocation for Intercluster References

When an intercluster reference is made, the context at the source cluster is suspended and cannot be allocated to other requests. The context must remain allocated both to allow error recovery and to allow Kmap operations with multiple intercluster references. To service an incoming request (from another cluster) also requires the allocation of a context. Thus , an intercluster reference requires that two contexts, one in each cluster, be allocated. Because there are only a small number of contexts, and that they are allocated sequentially and independently, there is a deadlock potential.

Consider 16 concurrent references, 8 from each cluster, to the alternate cluster. All contexts, in both Kmaps, would be allocated and none of the intercluster references could complete. Similar deadlock situations could arise, in more complex ways, in structures with

more than two clusters.

An initial approach to solving this deadlock problem might be to a timeout mechanism for the suspended contexts in the source cluster. It might be feasible to abort, and later restart, a request at the source Kmap. The aborted context would then be free to service incoming requests and so eliminate the deadlock. However, there is no obvious way to cancel the request pending at the destination cluster[1]. This dangling request might cause unpredictable results. For example, a side-effect producing operation (such as lock operation) might be performed twice. Also, data might be passed back to a context since allocated to an entirely separate transaction.

Another approach would be to provide a large number of contexts and somehow limit the number of incoming requests. We will see below that the number of incoming intercluster references can be limited to 112 for systems with up to 13,000 processors. With the present Kmap design it would be prohibitively expensive to support substantially more than the present 8 contexts[2].

The deadlock avoidance mechanism for contexts actually used in Cm* is quite simple. We note that to avoid deadlock it is only necessary to ensure that an incoming request is allocated a context within a bounded amount of time. As noted in the previous section, all intracluster references are guaranteed to complete without deadlock because they do not require allocation of a second context. Likewise, all incoming requests, once allocated a context in the destination cluster, are also guaranteed to complete in a bounded amount of time. It is possible to ensure that a context is eventually allocated to an incoming request. This is done by marking a context so that it cannot be used for references going out of the cluster. If this context is allocated for an outgoing reference, the request is aborted. (The

---

[1] This is an example of where the physically distributed structure and lack of central control makes conventional operating system deadlock avoidance schemes inapplicable.

[2] It is necessary to be able to concurrently read and write, at different addresses, the register files used for communication between the Kbus and the Pmap. At present, suitable IC's contain only 16 bits of data. To provide sufficient registers for 128 contexts would require on the order of 200 additional IC's. The present Pmap has a total of 155 IC's.

request will eventually get allocated to one of the other seven contexts[3].) This strategy ensures that at least one context is handling only within-cluster references and so will always eventually become available for requests from other clusters.

This simple mechanism guarantees freedom from deadlock over context allocation. It has only a very minor effect on performance: seven of the eight contexts are permitted to make intercluster references and the specially marked context is only allocated if all the others are busy.

3.5 Deadlock Over Buffer Allocation in the Lincs

In this section we consider the general problem of deadlock over buffer allocation, in a packet switched network. Much of this discussion will be applicable to any communication network dependent on intermediate buffering of information packets. The classic example of this is, of course, the ARPAnet. However, for concreteness the problem will be discussed in terms of communication between clusters in large Cm* networks. The Cm* clusters correspond directly to nodes in other networks, for example, the IMPs in the ARPAnet.

Figure 3-9 illustrates a deadlock situation in a simple four node network. The links or buses connecting the nodes form a cycle. This may be a subpart of a larger network.

For each arc leaving a node there is a single message buffer. In the example, messages are marked with their intended destinations. At node A, there is a message with destination C (diagonally opposite) waiting for transmission to intermediate node B. Similarly, there is a message at B, with destination D, waiting for transmission to intermediate node C. And likewise for nodes C and D. This leads to a cyclic resource dependency and deadlock.

3.5.1 Deadlock Prevention by Imposing a Partial Ordering

Most of the deadlock prevention schemes depend on imposing a partial ordering on the allocated resources (message buffers). If it can be shown that, for any message (or message class) passing through the network, buffers are always requested in a strictly monotonic order with respect to the partial ordering, then no cycles can exist. Hence the system would be deadlock free.

There are several important characteristics about individual messages that are, unless

---

[3] *It is possible that simply aborting the reference so that the request is re-arbitrated by the Kbus may lead to indefinite length delays in a saturated system. There is nothing to prevent the request being repeatedly allocated to the reserved context and then aborted. The Kmap microcode is being changed so that a queue of pending requests is maintained by the Pmap. The Pmap requests the next free context from the Kbus. The queued requests are processed in FIFO order.*

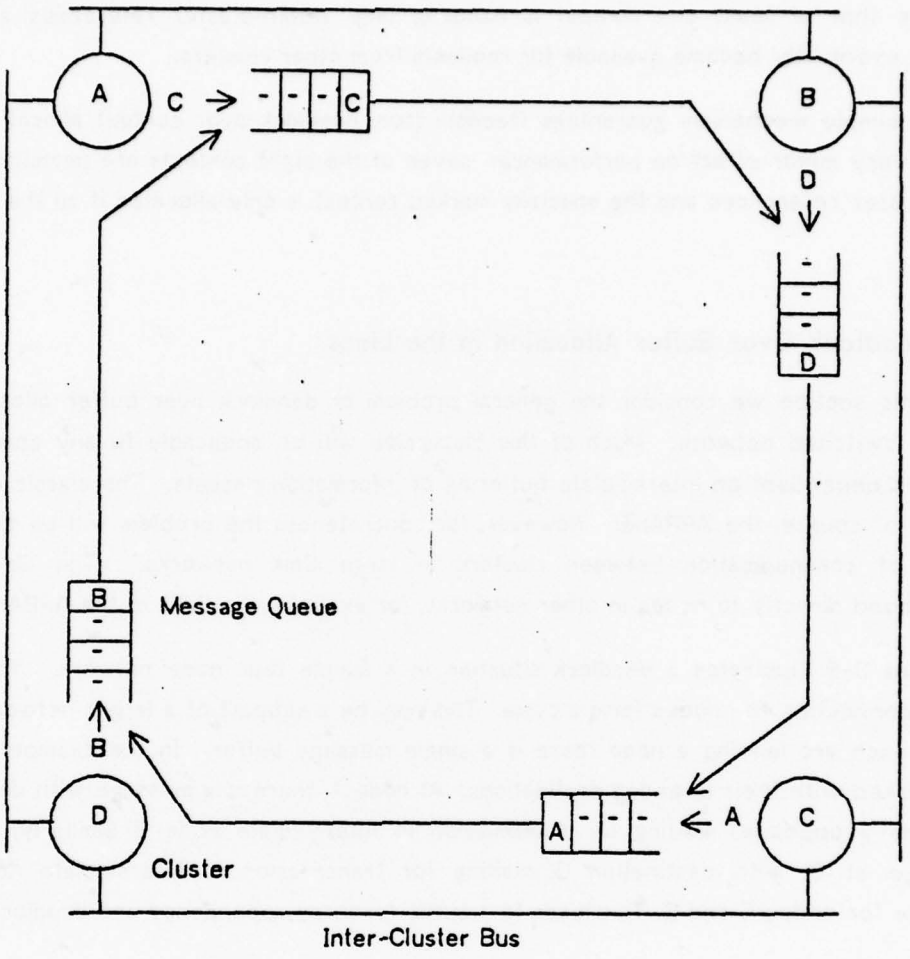


Figure 3-9: Deadlock over Buffer Allocation

otherwise stated, assumed in the following discussion:

1. The source and destination nodes for a message are distinct, i.e., the path of an individual message will not be a loop.

2. More generally, no reasonable message routing algorithm would cause a message path to include a loop, i.e., a message will not visit the same node twice[1].

[1] In systems with routing algorithms which respond dynamically to congestion and inter-node communication failures, for example, the ARPAnet, it may be possible for a message path to involve a loop in extraordinary circumstances. Some of the deadlock prevention schemes presented here will fail in those circumstances.

3. A more restrictive assumption is that a single path, between each pair of nodes, can be specified. Normally this would be a shortest distance (fewest intervening nodes) path. This places some limitation on the mechanisms used to respond to congestion or failure on a message path.

3.5.2 Division of Buffers into Destination Based Classes

If messages are classified on some criterion and only use buffers associated with their class then there can be no interaction between the classes and the deadlock potential of each class may be treated independently. Consider a spanning tree (preferably of minimum distance) from a Node A in an arbitrary network. The spanning tree, by definition, contains no loops and so defines a partial ordering of nodes leading to Node A. Thus if each Linc has an independent pool of message buffers for every potential destination cluster and incoming messages are allocated buffers according to their destination, the system is deadlock free.

This is essentially the mechanism proposed for the original computer module project [Fuller et al, 73]. Buffer classes were subdivided according to the destination segment. Only 16 segments were available simultaneously, via any inter-Cm bus. Even with this small number of buffer classes there are severe implementation problems. To correctly implement the mechanism, each node must cycle through each buffer class attempting to send any pending message. Frequently the successor node in the path will have to refuse a message, because its buffer class is full, while the buffers of most other buffer classes are empty. Trial designs by the author, showed that mechanism for cyclically attempting to transmit each message added substantially to complexity and reduced performance. No attempt has been made to estimate the performance degradation due to message refusal.

In the present Cm* structure, where a limit of 64 clusters is assumed in the microcode, 64 buffer classes would have to be provided. At 8 words per message, this would require 512 words of message buffer to provide only a single buffer per destination cluster.

3.5.3 Division of Buffers of Basis of Path Stage

Another basis on which to classify messages, and hence buffers, is the distance from the origin [Gunther, 75]. If the maximum path length (i.e., the maximum number of nodes visited) is P then buffers can be divided into P classes, C_1, C_2,..., C_P. A message begins at its source node by occupying a buffer from class C_1. When transmitted to the next node, it occupies a buffer class C_2. At each stage in its path, the message occupies a buffer from a successively higher buffer class until it reaches its destination. This imposes a partial ordering on buffer dependence and hence it is deadlock free.

In the worst case, this technique requires the same number of buffer classes as division on the basis of destination. However, for most network topologies the longest direct path does not include every node. Thus this technique usually requires fewer buffer classes than division on the basis of destination. It does require an additional field in the message which is incremented each time the message is forwarded.

3.5.4 Restrictions on Network Topology

It is possible to impose a partial ordering on message buffers by restricting the topology of the network. One obvious topology is a tree structure. Another structure is a square, cubic or higher dimension array. In a two dimensional array the horizontal and vertical buses might be called X and Y respectively, see Figure 3-10. Any node is reachable from any other node first travelling along an X bus and then (if necessary) along a Y bus. If this routing is enforced then a partial ordering is established between X buffers and Y buffers, and so the system is deadlock free. The major disadvantage of the regular array topology is that the failure of a single node can cause a whole column of clusters to inaccessible from certain parts of the network. Levy Raskin has suggested a class of structures which can tolerate a small number of node failures. In Figure 3-11 there are two paths between nodes, i.e. any single failure and most double failures will not prevent the continued deadlock free operation of the network. The limitation of this mechanism is obvious: restrictions are placed on the topology of the network. For small numbers of clusters this does not appear to be a burdensome restriction. In large networks, particularly when groups of semi-independent sub networks are connected, the regular structure may be an embarrassment. Difficulties also arise with messages which have to be redirected, eg. when a segment has moved from one cluster to another. The major advantage of this approach is that no special consideration is needed in the hardware design.

3.5.5 Deadlock Prevention through a Surplus of Resources

Circular resource dependency can also be avoided by ensuring that the pool of resources can never be exhausted. The maximum number of extant intercluster messages in a Cm* network is $7 * N_{clusters}$. (Only 7 contexts from each Pmap are permitted to make intercluster references.) For a network of 64 clusters this amounts to 448 extant messages, or a total of 3.5 K words of message storage. For a cycle to exist at least two nodes must be involved. Therefore a Linc would need storage for only half the maximum possible number of messages. Thus if each Linc had 2K words of storage (32 1K bipolar RAM chips) then the system would be deadlock free, there would be no topological restrictions and there would be no wasteful reservation of buffer space. With 1 K of storage in each Linc up to 256 extant messages

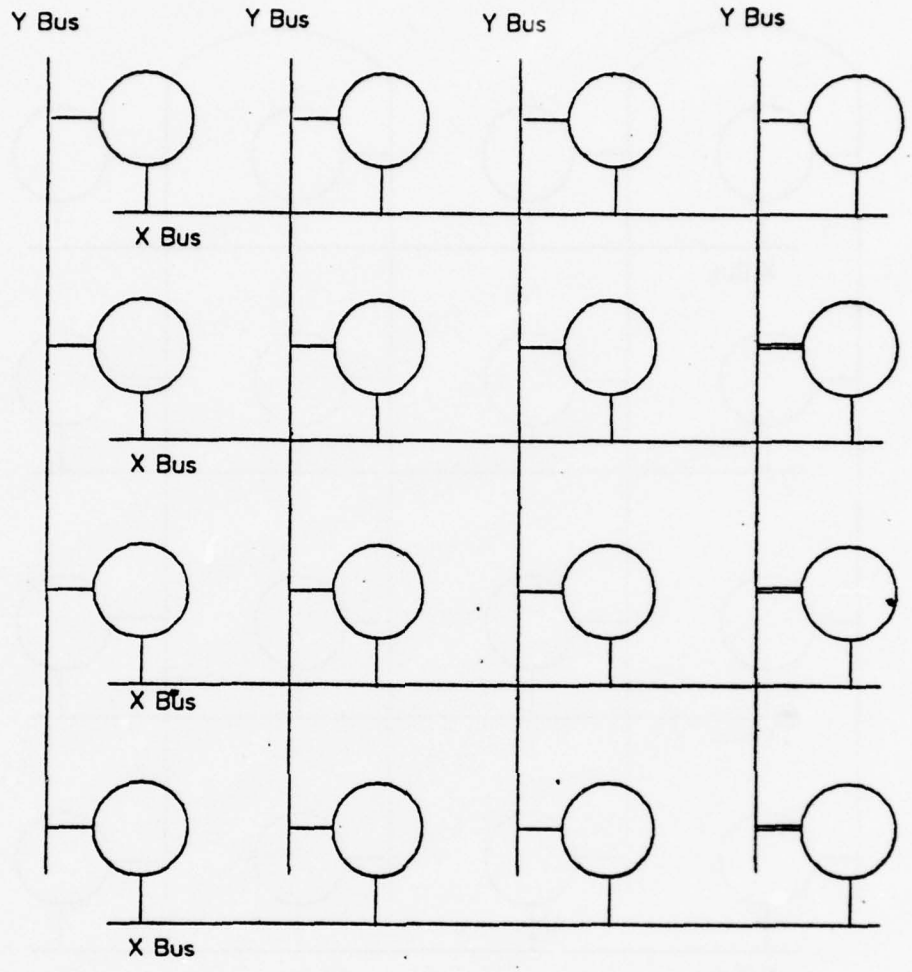


Figure 3-10: Regular Array. Any node can be reach via first an X then a Y Bus

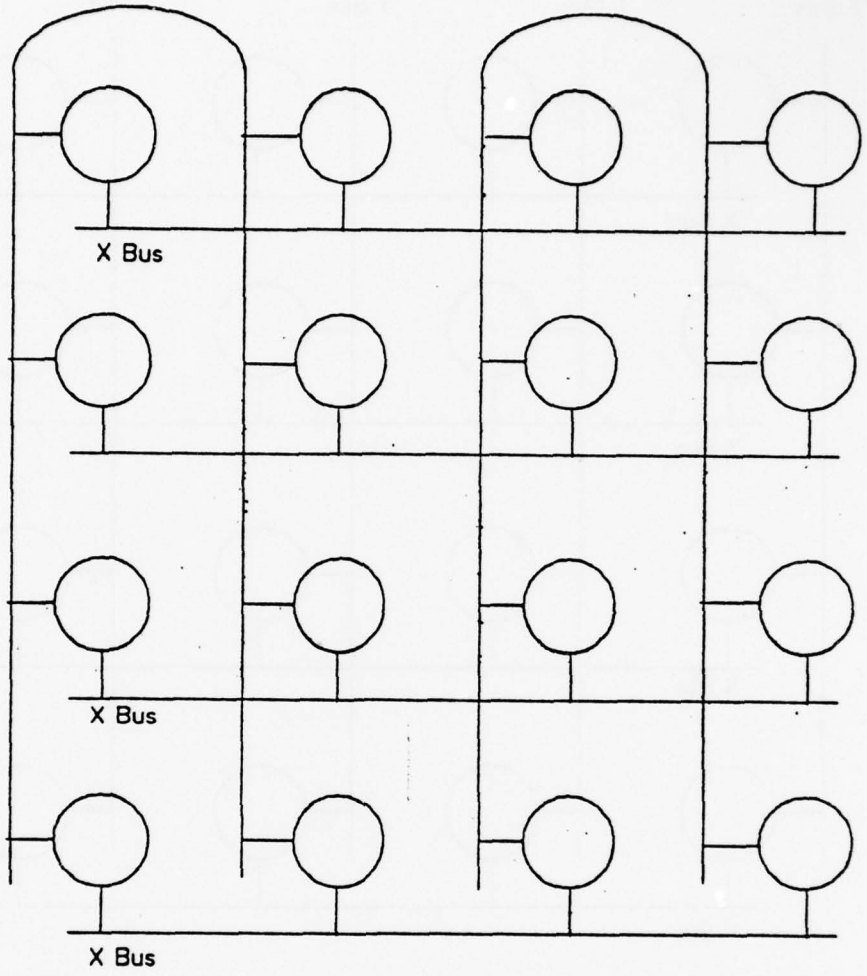


Figure 3-11: Two Deadlock Free Paths between each Node

could be accommodated. This corresponds to limiting the network to 36 clusters or in a 64 cluster system allowing a maximum of 4 outstanding intercluster requests.

For networks where the total number of extant requests can be controlled and the cost of an individual buffer is small, this is a very practical approach. In systems where individual messages or packets may be very large, the memory costs of providing for the worst case situation may be prohibitive. In Cm*, an 8 word packet size appears to be ample. The

present Cm* Lincs are implemented with 1K 16 bit words of fast buffer storage, allowing 120 message buffers in a single pool. (Eight buffers are reserved for preparation of outgoing messages for each cluster.) This guarantees freedom from deadlock for up to a 36 cluster (maximum of 504 processors) system. With 1978 technology, a system four times this size could be accommodated with the same number of ICs.

An extension to this mechanism would be to provide only sufficient fast buffer storage to handle expected traffic. Buffer space exhaustion would be relieved by reliance on cheaper, slower storage. In the case of Cm*, the overflow memory would be primary memory within the cluster. In the ARPAnet it might be secondary storage on a host connected to an Imp.

3.5.6 Special Handling of Reply Messages

In Cm* there is an explicit acknowledgement for every packet sent between clusters. In the case of a read operation it is the requested data. Write operations are also acknowledged for reliability reasons. The source context is suspended pending a reply message.

Reply messages are queued independently from incoming requests by the Linc and are given precedence. This is so that the suspended contexts (and associated processors) can be freed as quickly as possible. Note that this is a performance argument and is not a question of deadlock. The mechanism described in Section 3.4.4 ensures that a context is eventually allocated to every incoming message.

3.6 Limitations on the Size of Cm*?

In Section 2.2 we argued that Cm* was extensible along the dimensions of processing power, memory capacity and intercommunication bandwidth. The issue of communication bandwidth is covered in that section; here we will examine actual limits on the total number of processors and amount of primary memory.

In the current hardware implementation the following restrictions apply:

1. Limit of 248K bytes of primary memory per Cm. (This is due to the 18 bit address path in the Slocal and on the LSI-11 bus.)

2. Limit of 13 Cms (processor with memory) per cluster. (The number of Cms is limited by the arbitration logic in the Kbus and the number of address lines on the Map bus.)

3. Limit of 64 clusters per intercluster bus. (The address recognition logic in the Lincs inspects only 6 bits. A practical limit may be lower due to bus length and loading.)

4. Limit of 128 message buffers per Linc, allocated from a common pool. With a maximum of seven outstanding intercluster messages per cluster this corresponds to the capacity of 18 clusters.

The present microcode limits the total number of clusters to 64, which corresponds to 896 processors. The microcode also limits the virtual address space to 2^{28} bytes which gives 1024 processors with a full complement of memory, or 2048 processors for 128K bytes each (as planned for the 50 Cm system).

As noted in Section 3.5.5, deadlock considerations-- for the worst case--limit the structure to 36 clusters or 504 Cms. By simply limiting the number of outstanding messages per cluster to four, by a change in the microcode, this could be expanded to 64 clusters or 896 processors. (Limiting the number of outstanding intercluster requests would probably have a very slight effect on performance.)

There is a spectrum of restrictions on the network topology which allows larger structures without deadlock. An 18 * 18 matrix leads to 324 clusters or 4536 Cms. By also restricting the number of outstanding messages to four, a 32 * 32 cluster matrix would be deadlock free. This provides 1024 clusters and 13,336 processors. To make use of this hardware structure, the microcode would have to be changed to support the large address space. This structure could contain approximately 4000M bytes of primary memory and have a raw performance of 2,400 MIPs. In practice, construction of a system of this size would be limited by space and cooling requirements[1] and the software to effectively use the processors.

[1] With one cluster per cabinet, and rows of cabinets spaced 6 ft. apart, the system would fit in a room 200 ft. by 64 ft. Built with 1978 technology it could be perhaps ten times smaller.

4. Aspects of Cm* Implementation and Performance

In previous Chapters we have described the basic structure and operation of Cm* and examined techniques for preventing deadlock in the switching structure. In this chapter we will examine Cm* in more detail. We begin with an examination of the overall performance of the system. This is done both through measured results and a calibrated analytic model. Cm* has three levels of buses: the LSI-11 memory bus (Q Bus), the Map Bus and the Intercluster bus. The Q Bus is a standard commercial product. The protocol and design motivations of the Map Bus and Intercluster bus are examined in this chapter. Up to this point there has been little discussion of the design of the processors for Cm*. We will examine the modifications necessary to the LSI-11 processors so that they can operate correctly in the packet-switched structure of Cm*. In addition, we will discuss the other modifications necessary to make an inexpensive microprocessor acceptable in an operating system environment.

4.1 The Performance of Cm*

The viability of Cm*, as a switching structure which can effectively support a large number of processors, depends primarily on two factors. Firstly, the ability to implement cheaply and quickly, references to non-local memory. Secondly, the existence of applications which suitably decompose for a multiprocessor and achieve a sufficiently high locality of reference.

In this thesis we are not concerned with the decomposition of algorithms, but we are concerned with the issue of locality of reference. We can express the effective, or average, relative memory reference rate of a processor as follows:

$$(\text{Hit ratio} * t_{local} + \text{Miss Ratio} * t_{non\text{-}local})/t_{local}$$

The Hit ratio is the fraction of all references which are to local memory.

The Miss ratio = 1 - Hit ratio.

t_{local} is the average time between references to memory local to the processor.

$t_{non\text{-}local}$ is the average time between reference to memory local to some other processor in the Cm* network.

This simple formula demonstrates that overall performance depends on the relative frequency of local references and the relative cost of non-local references. As the ratio of non-local references increases (Miss Ratio), processor utilization decreases. This means that to maintain reasonable processor utilization, we must limit the percentage of non-local

references[1]. This in turn limits the nature of the algorithm decompositions which can be effectively run on Cm*.

The range of applicability and ease of decomposition is enhanced by making the delay associated with a non-local reference as small as possible. This was an important objective in the engineering design of Cm*. However, maximum performance was sought within the following constraints:

- Limitations imposed by the LSI-11s.
- The use of schottky TTL logic rather than ECL.
- Overall conservative design--particularly with bus structures and mechanical packaging.
- Reasonable cost.

4.1.1 Single Cluster Performance

Figure 4-1 shows the measured relative performance of eight processors in a single cluster as a function of the hit ratio to local memory[1]. This shows that relative performance, or processor utilization, does not drop below 50% until the hit ratio drops to 55%. At a hit ratio of 70%, the system is performing memory references at about 63% of the rate of eight independent LSI-11s. With a hit ratio of 90%, i.e., 1 out of 10 references are to memory elsewhere in the cluster, the processors are being used at about 82% of efficiency. At 95% hit ratio, 90% of the potential processing power is available.

On the same graph (Figure 4-1), the performance of a one processor cluster is also plotted. The relatively small difference in performance between the single and eight processor case shows that a high degree of concurrency in both the Map Bus and Kmap has been achieved. At least with up to eight processors, the limit of available measurements, contention for the shared resources of the Map Bus and Pmap have only slightly affected performance.

The third plot on the graph shows the affect of contention for a single target memory. This illustrates the situation where all non-local references are to a single global data structure, held within a single physical memory. With 8 Cms and a local hit ratio of 90%, there is little advantage in physically distributing the global. However, below a hit ratio of 75% the

---

[1] For example, to limit processor degradation to execution at 66% of the stand alone performance, we can simply derive:

$(t_{non\text{-}local}/t_{local} - 1) \cdot \text{Miss ratio} \leq 0.5$

[1] This figure is reproduced from [Swan, 77].

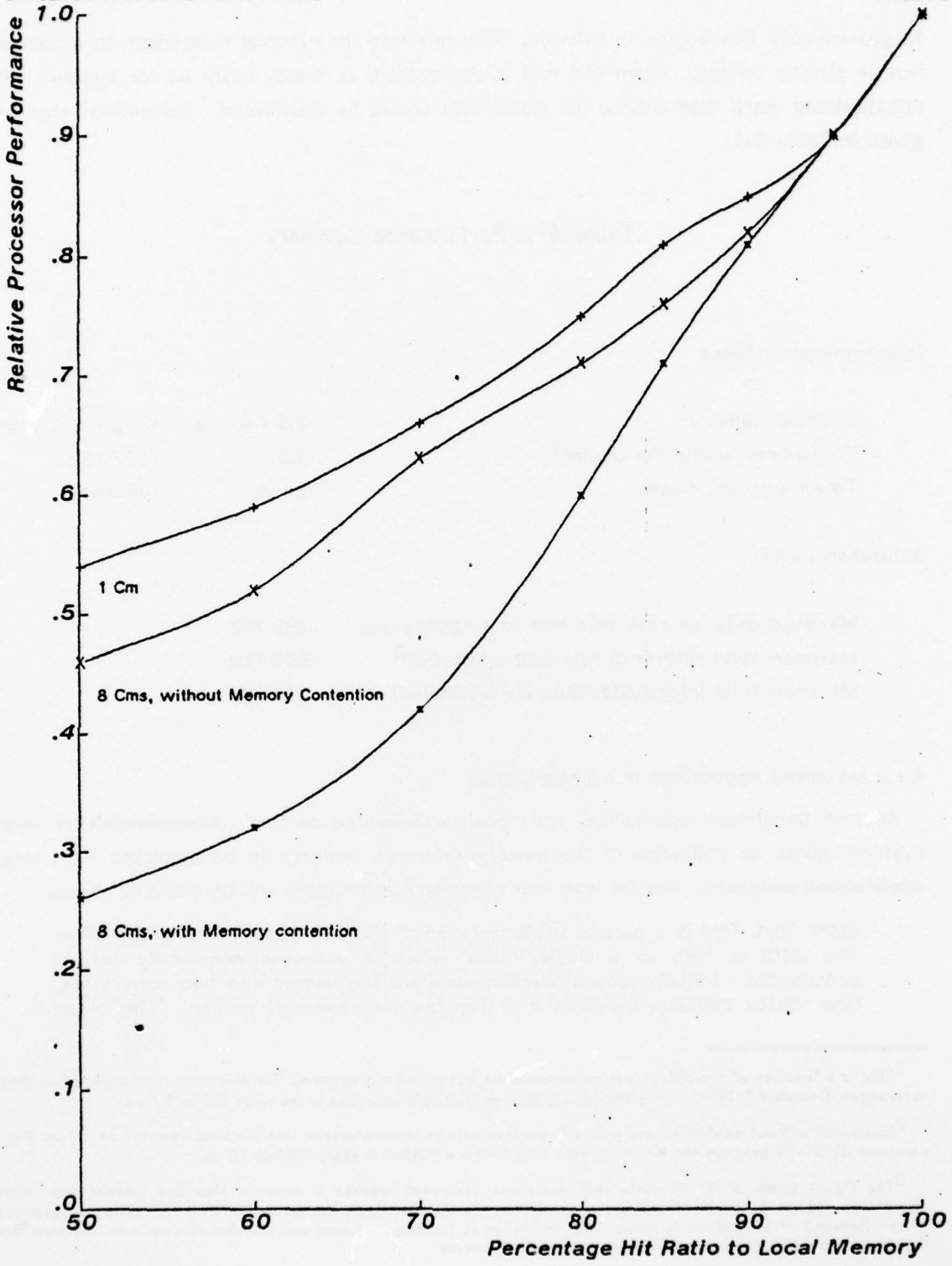


Figure 4-1: Relative Processor Performance, or Utilization

target memory bus begins to saturate. The data rate for external references to a memory bus is strictly limited. When this limit is approached, as it may easly be for systems with substantially more than 8 Cms, the global data should be distributed. Saturation rates are given in Table 4-1.

Table 4-1: Performance Summary

Inter-reference Times

To local memory[1]	2.9 - 4.0 us	(345 - 250 KHz)
To memory within the cluster[2]	9.3	(107 KHz)
To an adjacent cluster	24 us	(42 KHz)

Saturation Levels

Maximum external reference rate to a memory bus	250 KHz
Maximum total reference rate over a Map Bus[3]	500 KHz
Maximum total Intercluster Bus rate (estimated)	312 KHz

4.1.2 Measured Applications in a Single Cluster

Several benchmark applications have been implemented on Cm*. Measurement of these systems gives an indication of the memory reference patterns to be expected from large applications programs. Results from four benchmark applications will be discussed here:

- Quick Sort. This is a parallel implementation of the well known sorting algorithm. The data is held in a single vector which is accessed non-locally by all processors. Initially, one processor partitions the vector into two subvectors. One vector contains elements less than the (approximate) median. The other

---

[1]This is a function of the LSI-11 performance and the instruction mix measured. The Slocal does not slow down these references. Compiled BLISS-11 programs usually have an inter-reference time in the range 3.3 to 3.7 us.

[2]Measured without contention and with a simple loop with an inter-reference time (to local memory) of 2.9 us. For a compiled BLISS-11 program the inter-reference time (within a cluster) is approximately 10 us.

[3]The figure given is for complete read operations. Each read requires 3 separate Map Bus transactions. Write operations require a fourth Map Bus transaction to pass the data. Thus the saturation level for write operations is lower. Because of overlapping between DMA arbitration at the target memory and transfer of data over the Map Bus, individual write operations take the same elapsed time as reads.

contains elements equal to, or greater than, the medium. Pointers to one subvector are placed in a common pool, the processor then proceeds to partition the other subvector. All processors operate identically. They take pointers to a vector from the shared pool, partition it into two subvectors, return one subvector to the pool, and further partition the other subvector. The sort is complete when no vectors remain in the common pool.

- **Partial Differential Equations.** The method of finite differences is used to solve a Partial Differential Equation (PDE). Various algorithms developed by Gerard Baudet [Baudet,78] were measured. All methods operate iteratively on a global matrix. The results presented are for an innovative technique which requires minimal synchronization between cooperating processors. (See [Baudet,78] and [Raskin,77] for further information.)

- **Integer Programming.** This is a search problem requiring extensive arithmetic and logical operations. The particular domain chosen was airline crew scheduling [Balas,76]. Given a set of flight logs (from city A to city B at time T) and a number of airline crews, find a minimum cost allocation of crews to flight legs. (See [Raskin, 77] for further information.)

- **Speech Recognition.** This is a multiprocessor implementation of the Harpy speech recognition system [Lowerre, 76]. Speech input is preprocessed and segmented by another computer. The implementation on Cm* requires searching, in parallel, a few "best paths" in a state transition network. Probabilities in the state transition network, which represents the knowledge base of the system, are dynamically updated.

These applications were not chosen because they had a natural decomposition for the hierarchical switching structure of Cm*. On the contrary, an attempt was made to find the limits of suitability of the structure. The sorting application was chosen because it is highly data intensive. A single vector of integer items is sorted in place. This vector is shared by all processors. All references to the data are non-local and so incur approximately a factor of 3 slow down[1]. The speech recognition and PDE applications were transferred directly from C.mmp. They were initially decomposed for a multiprocessor with uniform access time to memory and were not restructured for Cm*.

The performance of these algorithms, as a function of the number of processors available, is shown in Figure 2-15. The integer programming and Partial Differential Equation applications show close to linear speedup as processors are added. Speech recognition and sorting show reasonable speedup for up to 3 or 4 processors. Beyond that, there is little advantage in adding processors. The trailing-off of performance improvement must be due, either to the nature of the algorithm, or to limitations of the switching structure used to access the shared data. Results from other implementations, and the data given below, shows

[1] The shared data is in the local memory of one of the participating processors. However, for consistency, this processor also made references to the data via the Kmap.

that it is inherent in the applications and is not a consequence of delays in the access to non-local data.

4.1.3 The Affect of Switch Delay on Applications

The memory references of each application can be classified into four groups: code, stack, owns and globals. The stack and owns are private to a process. The code, because it is read-only, may be either shared (to save memory) or duplicated for each process (to avoid non-local references). The globals must be common to all processes, hence they must (usually) be accessed non-locally. The percentage of memory references to each of these classes is shown in Table 4-2 for the four benchmarks.

Table 4-2 shows that 70% to 80% of all memory references are to code. This indicates that it is essential to insure that frequently used code sequences are available locally. References to the stack and owns constitute from 10% to 25% of all references. Thus it is advisable to allocate the private data of a process local to the processor executing the process. The percentage of references to global data varies from about 1% to 15%. This data cannot be duplicated, local to each processor. References to it must be non-local.

Table 4-2: Benchmark Data

Application	Code	Stack	Owns	Global	Local Hit Ratio	Degradation due to Switch
Integer Programming	71%	24%	4%	1%	99%	2%
PDE's	82%	11%	4%	3%	97%	6%
Quick Sort	71%	12%	7%	10%	90%	17%
Speech Recognition	75%	10%	-	15	85%	22%

The total performance degradation, due to slower non-local references, is also shown in

Table 4-2. These figures assume that code, stack and owns are held local to each processor. Only global data is accessed remotely. The amount of performance degradation is calculated from the measured switch performance data given in Section 4.1.1

Table 4-2 shows that, for the PDE and Integer Programming applications, no more than 5% of performance is lost due to the hierarchical switching structure of Cm*. The largest performance loss shown is for the Harpy speech recognition system, 22%. The percentage of non-local references could be reduced by copying all, or part, of the read-only global data. It is likely that the local hit ratio for both Harpy and Quick Sort could be significantly improved through restructuring of the programs. However, as the maximum potential performance improvement is relatively small, the programming effort may not be justified. For the benchmarks used, the Cm* structure has close to the same performance as a multiprocessor with a full crosspoint switch.

4.2 Multicluster Performance

One of the major advantages claimed for Cm* is its extensibility to large numbers of processors. Measured data from applications is only available from the present 10 processor system. However, simulation results [Raskin, 78] are available for systems with up to 48 processors. In this section, we will examine the performance of a 4 cluster, 48 processor Cm* system. The structure is shown in Figure 4-2. The clusters, with 12 Cm's each, are connected by a single Intercluster bus.

For any application running on the system, there are two parameters relevant to the performance degradation due to the switching structure:

1. **Local Hit Ratio.** This is the percentage of all references which are to local memory. In single cluster systems, this is the only important performance parameter.

2. **Intercluster Ratio.** This is the fraction of non-local references which are passed outside the cluster. In the simulation model, all references outside the cluster are assumed to be uniformly distributed to the other 3 clusters.

Figure 4-3 shows the performance of a 48 processor system for a representative range of parameter values. The uppermost curve has the Intercluster Ratio = 0. That is, all references remain within the cluster. Thus, the curve simply shows the relative performance of 4, non-interacting, clusters as a function of the hit ratio to local memory. This is an upper bound on multicluster performance. The lowest curve has the Intercluster Ratio = 0.75. That is, 3 out of 4 non-local references are passed out of the cluster. This reflects a situation where references to global data are uniformly distributed across the system. There is no locality of reference at the cluster level. This can be considered a lower bound on

performance for a reasonable, multicluster application decomposition.

The lower bound curve shows that, without locality of reference at the cluster level, processor utilization is reduced to about 85% with a local Hit Ratio of 97.5%. Thus the PDE and Integer programming benchmarks can be expected to operate effectively, without restructuring, on a 48 processor system. However, processor utilization is reduced to about 58% with a 95% Local Hit Ratio. With a 90% Local Hit Ratio, processor utilization is reduced to 27%.

The performance of the 4 cluster system is greatly enhanced if the application can be decomposed to give locality at the cluster level. With a 90% Local Hit Ratio and a 0.1 Intercluster Ratio (i.e. 1 reference in 10 is non-local, of these 1 in 10 is to another cluster) average processor utilization is about 77%. This is reduced by only 4% from processor utilization with a 90% Local Hit Ratio and no intercluster references. Figure 4-3 also shows performance for an Intercluster Ratio of 0.25 (25% of non-local references are passed to another cluster) and 0.5 (50% of non-local references are passed to another cluster).

4.2.1 A Test of the Cluster Concept

An alternative way to utilize 48 processors would be to have single cluster with 48 processors, rather than 4 clusters with 12 processors each, as discussed above. This is not possible with the present hardware implementation, but would be quite feasible. Figure 4-4 shows the (simulated) performance of a single cluster with 48 processors. This performance curve is super-imposed on the curves from Figure 4-3. The single cluster is superior when references to global data are uniformly spread across all memories. When cluster level locality is present (Intercluster Ratio < 0.5) the decomposition into 4 clusters is superior. The advantages of the decomposition into clusters will increase for systems with more processors.

4.2.2 Summary of Results for 48 Processor System

We have examined the performance is a specific 48 processor, 4 cluster system. Closely calibrated simulation results show:

- For applications with a Local Hit Ratio better than 97%, such as the PDE and Integer Programming benchmarks, better than 75% processor utilization is obtained without need for cluster level locality. The programmer need not seek a problem decomposition corresponding to cluster boundaries.

- For almost all Local Hit Ratios, there is a significant performance benefit if the application is decomposed to give locality of reference at the cluster level. If 90% of references are local (as in Quick Sort) and 90% of the non-local references remain within the cluster, the processor utilization is 77%.

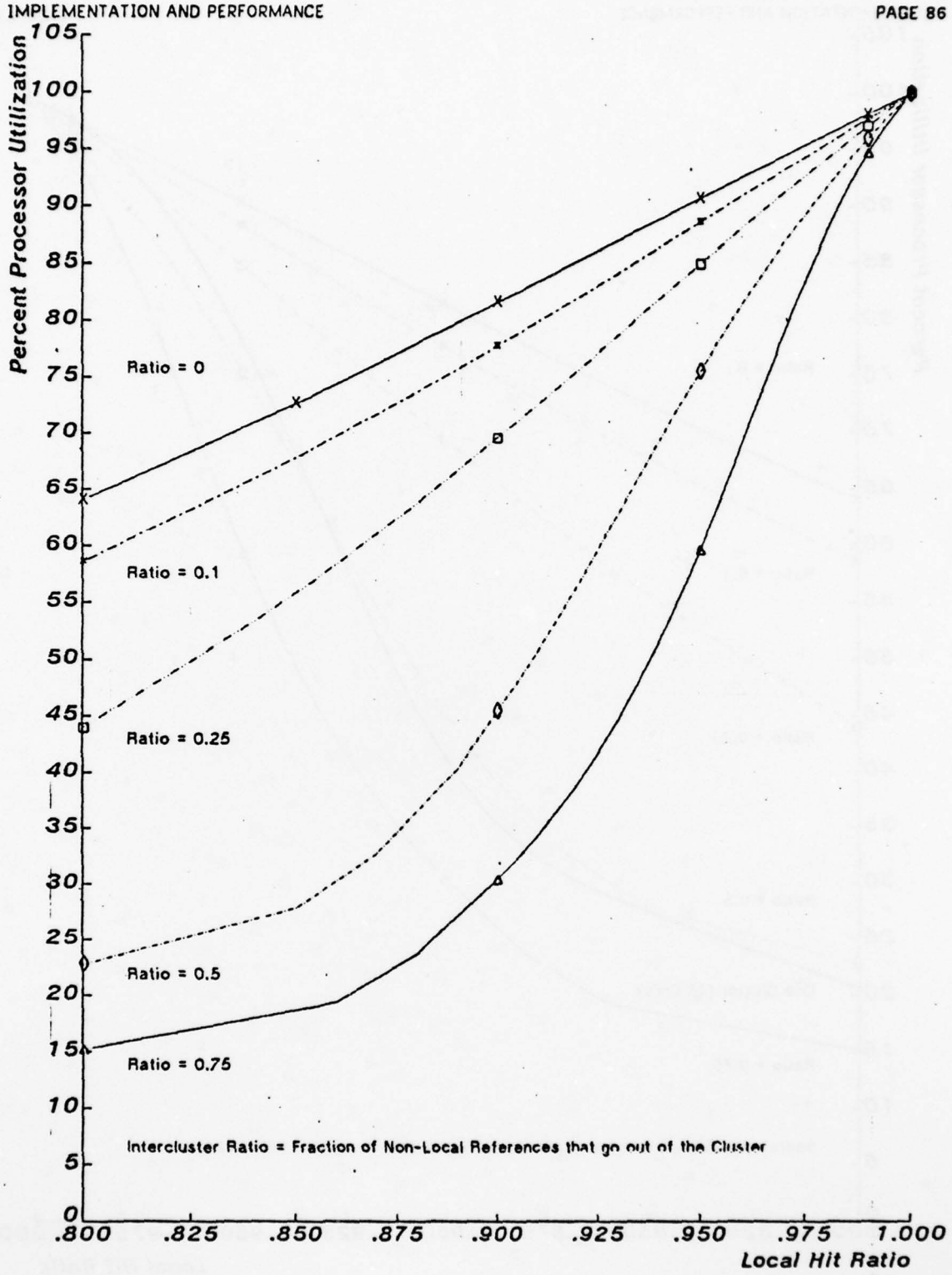


Figure 4-3: Performance of a 4 Cluster, 48 Processor System

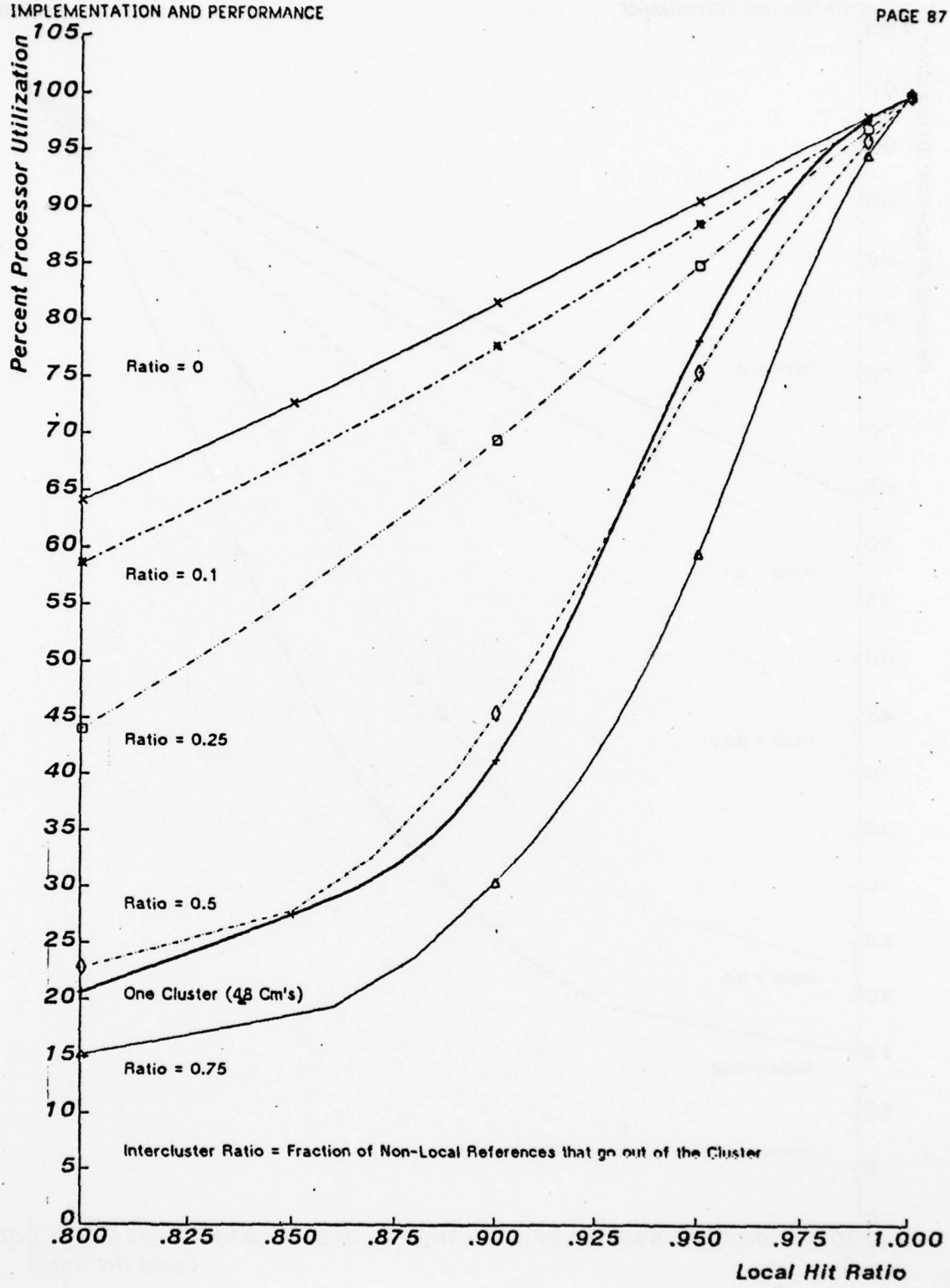


Figure 4-4: 48 Processors Distributed across 1 Cluster, or across 4 Clusters

- Where cluster level locality is found, 4 separate clusters is superior to a single cluster with 48 processors. If little cluster locality exists, a single large cluster offers better performance.

The overall system performance, which we have just examined, is dependent on many individual system components. The design and performance of some of these components are discussed in the following sections.

4.3 The Map Bus Protocol

The Map Bus is the bus which allows communication between the Computer Modules and the Kmap within a cluster. The Map Bus protocol can have a large impact on overall system performance because it must carry all non-local references by up to 14 processors. The Map Bus protocol may also have a significant impact on system cost, because every Cm must include a Map Bus interface.

4.3.1 The Function of the Map Bus

The Map Bus is not a general purpose bus in the sense of the DEC Unibus, etc. It is designed to have high performance and low cost for a specific function, the transfer of small packets of information (about 20 bits) between units within the same cabinet.

Information packets (address or data) are mostly transferred between a particular Computer Module and the Kmap. However, to avoid double bus transactions, information can be transferred directly between any two Computer Modules. This is used for the data portion of a write operation (the Pmap must translate the address portion but need not see the data portion) and for the return of data (for a within cluster read) or return of an acknowledgement (for a within cluster write).

4.3.2 Request Types

A transaction on the Map Bus can be invoked by a Cm for two independent purposes:

1. A **Service Request** indicates that a processor has begun a non-local memory reference. That is, the processor has generated a 16 bit address which must be translated by the Pmap and routed to the appropriate destination.

2. A **Return Request** indicates that a reference to primary memory within that Cm, invoked by the Kmap, has been completed. That is, data (or acknowledgement) is ready to be transferred to the processor originating the request.

These two request types, corresponding to the two major functional components of a Cm--the processor and the memory--are independent and can occur concurrently. It is important to give precedence to Return Requests. Acknowledgement, by a Map Bus transfer, of a Return Request releases two important resources: (1) the context allocated to the transaction, and (2) the target memory[1]. Until service begins, a pending Service Request holds no resources other than the suspended processor.

4.3.3 Restrictions on the Map Bus Topology

To maximize performance, limit cost and simplify design two related restrictions are placed on the Map Bus.

1. A moderate upper limit (14) is placed on the number of Cm's per Map Bus.
2. The Cm's are packaged close together, limiting the length of the Map Bus cable.

These restrictions closely relate to the notion of locality within a cluster, and the realization that bus bandwidth will limit the useful number of Cm's per cluster.

4.3.4 Map Bus Arbitration

The restriction to an upper limit of 14 Cm's per Map Bus permits use of a central, parallel arbitration scheme with independent request lines from each Cm. This central arbitration is performed in the Kbus. The individual requests lines (rather than a common request line with a daisy chained grant) leads to reduced logic in the Slocals and to a number of functional and performance advantages (see Section 4.3.7).

There are a total of 28 request lines, 14 service Request and 14 Return Requests. To reduce the cabling and connector costs, the request lines are divided into two groups, each serving 7 Cm's. Each Slocal has connections for only 14 request lines. In a maximum configuration, the Kbus must be physically in the middle of the Map Bus with up to seven Cm's on each side.

4.3.5 Map Bus Signals and Data Formats

Table 4-3 describes the signal and data lines used for the Map Bus. The Map bus is operated synchronously, with all timing and control signals generated by the Kbus. This gives

---

[1]Note, the DMA portion of the Slocal can handle only a single reference. The Slocal is marked "busy" in a table in the Kbus until the transaction is complete. However, the processor and other devices on the target memory bus can make memory references while the Slocal is waiting for the Map Bus.

AD-A060 494 CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
THE SWITCHING STRUCTURE AND ADDRESSING ARCHITECTURE OF AN EXTEN--ETC(U)
AUG 78 R J SWAN F44620-73-C-0074
UNCLASSIFIED CMU-CS-78-138 NL

2 OF 3
AD
A060494

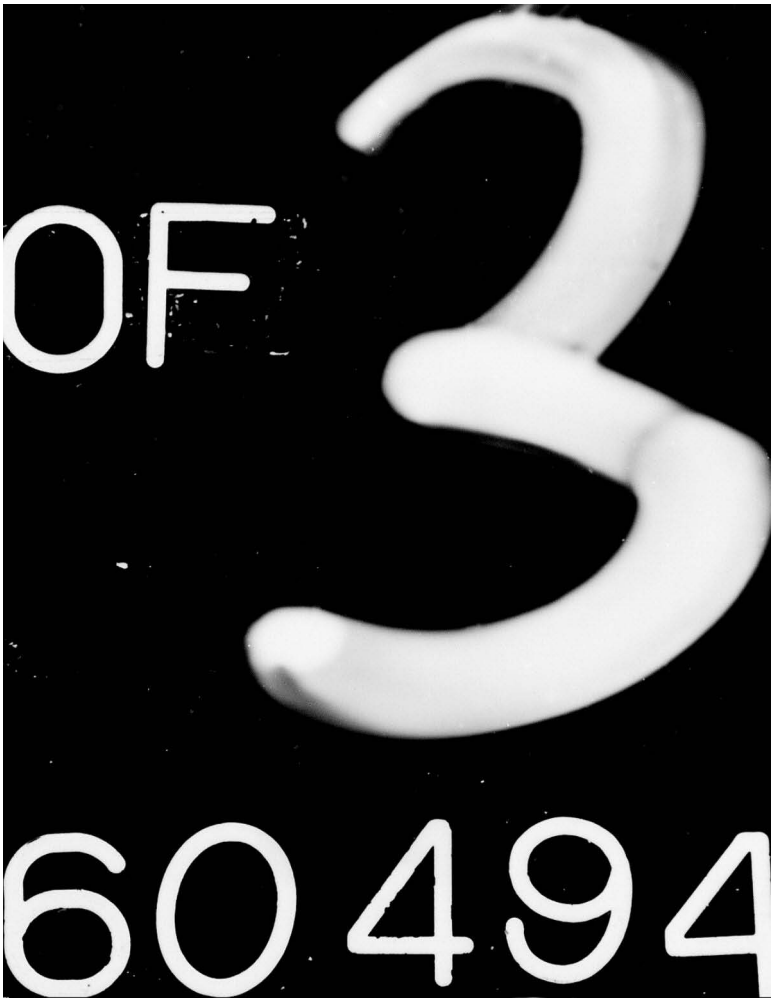

OF
3
60494

Table 4-3: Map Bus Signals

Service Request	Asserted when a processor generates a non-local memory reference. I.e., when it requires service by the Kmap. The line remains asserted until the operation is completed successfully, or an error trap is forced, or the Slocal times out and the processor takes a time out trap.
Return Request	Asserted when the Slocal has completed a memory reference requested by the Kmap. It is released when the data (or acknowledgment) is read--usually in a direct transfer over the Map Bus to a requesting processor.
Source Addr<3:0>	The address of the Cm which will place information on the Map Bus for this transaction.
Source Strobe	The strobe signal which indicates that the Source Address is valid and causes the selected Slocal to enable its information onto the Map Bus. Note, no strobe is generated if the Kmap is the information source.
Destination Addr<3:0>	The address of the Cm which will latch information from the Map Bus.
Destination Strobe	The strobe signal which indicates that the Destination address is valid; the information on the Map Bus should be latched on the trailing edge of the strobe. No strobe is generated if the destination is the Kmap.
Information<19:0>	These lines carry the address, data and control signals between the Cm's and the Kmap. (See Figure 4-5 for the data formats.)
Parity LB, HB, CTR	Parity on the Information lines. It is generated by the Source (Slocal or Kbus) and checked at the Destination (Slocal or Kbus).
Type and Phase	These signals define the nature of the Map Bus transaction and are used to directly address the appropriate source and destination registers. (See elsewhere for the encodings.)
Map Bus Error	This is an error line common to all Slocals. It is asserted by any Slocal that receives an address or data word with bad parity on the Map Bus, or when a Parity or non existent memory error occurs during a reference by the Slocal to the LSI-11 bus. A more specific error indication is contained in the control information which can be read from the Slocal. Reading this information releases the error line.
Map Bus Flag	This is a special error line used only when a processor is passed data with bad parity after a non-local read operation. This is treated as a special case because it is essential that the context (in the Pmap) associated with this transaction not be released until valid data is returned to the waiting processor. (See text.) A second, independent function of this line is to allow implementation of unit enlocked write operations. If asserted (by the Kbus) during a read of the data generated by the processor, the processor will continue execution.
Map Bus Init	This is asserted by the Kbus, under microprogram control from the Pmap, to initialize all Slocals on the Map Bus.

fast performance while minimizing the control logic in the Slocals. Each Map Bus interface is essentially three registers which can be enabled onto the Map Bus and three registers which can be loaded from the Map Bus. (See Figure 3-4.) These registers are directly addressed by the Map Bus control signals, **Type** and **Phase**:

Table 4-4: Registers on the Map Bus

Type	Phase	Source	Destination
0	0	Address from Processor	Address to DMA
0	1	Data from Processor	Data to DMA
1	0	Old DMA Address[1]	Not Used
1	1	Data from DMA	Data to Processor

Used as a single address bus, i.e., for a transfer between a selected Cm and the Kmap, the Kbus may read or write any register directly. When used as a two address bus, for a direct transfer between a selected Source Cm and a selected Destination Cm, the registers can only be accessed in pairs. The register addresses for Source and Destination are appropriately matched. For example, for a write operation the data may be transferred by the Kbus asserting the source and destination addresses, **Type** = 0 and **Phase** = 1; asserting **Source Strobe** enables the data from the processor onto the Map Bus and asserting **Destination Strobe** causes it to be latched in the "Data to DMA" register. The formats for information transfers over the Map Bus are shown in Figure 4-5.

4.3.6 Error Detection and Retrys on the Map Bus

All transactions on the Map Bus have parity, which is generated at the source and checked at the destination. However, the Map Bus is operated synchronously--there are no interlocked acknowledgements that, conventionally, would be used to signal transmission errors. The use of a fully interlocked bus structure would increase the Slocal complexity and probably substantially reduce performance. There are three cases to consider when handling Map Bus errors:

1. A transfer from a Cm to the Kbus. The Kbus is checking the received parity and so can respond reasonably. On the assumption that it is a transient error, the

[1] Used for hardware diagnostics only.

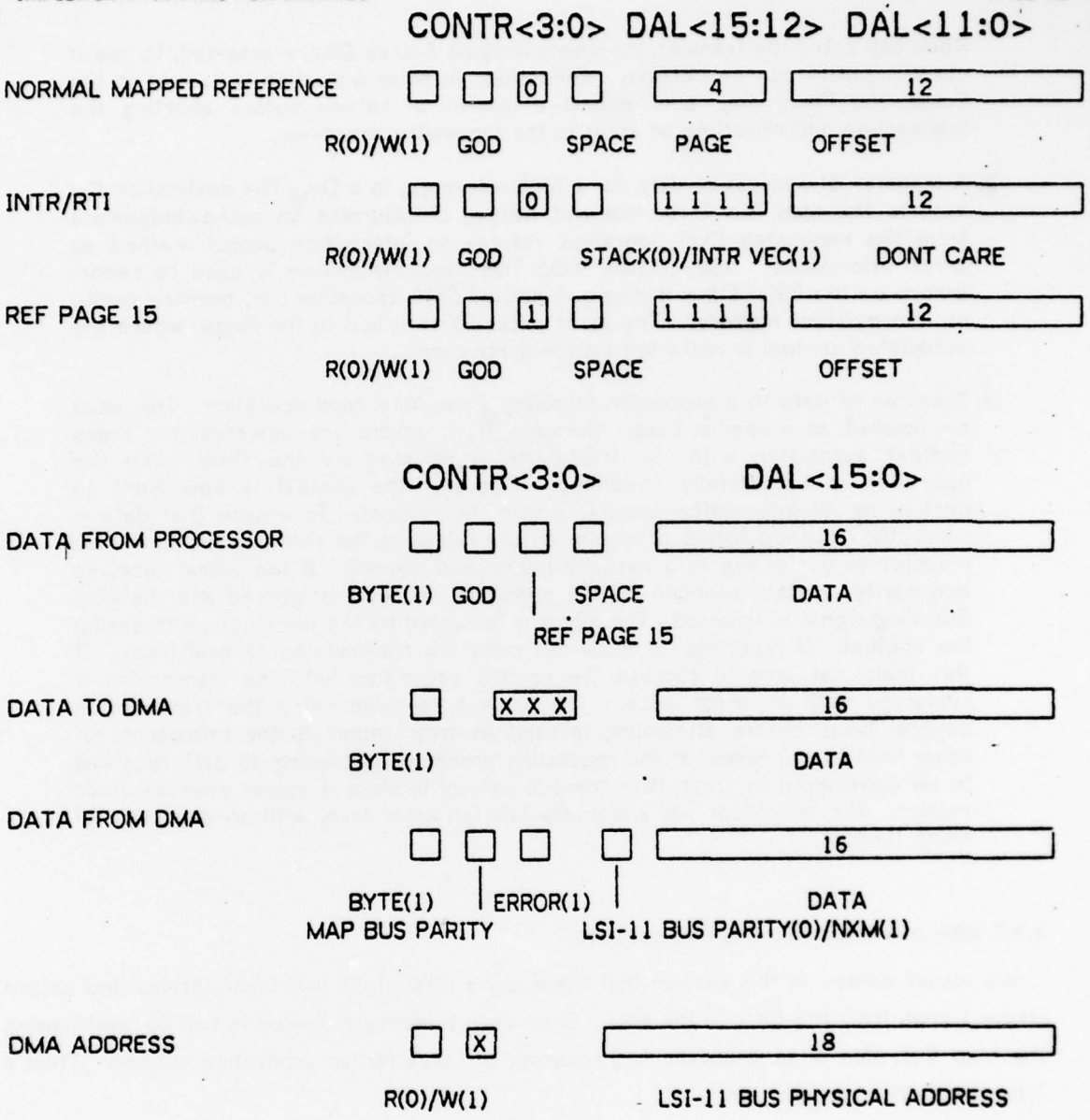


Figure 4-5: Format of Transactions on the Map Bus

Kbus can retry the transfer, by simply keeping Source Strobe asserted, to see if correct parity can be obtained. Otherwise, an error indication is passed to the Pmap; the Pmap may also request a number of retries before aborting the transaction and reporting an error to the requesting processor.

2. A transfer of address or data, for a DMA reference, to a Cm. The destination Cm asserts the Map Bus **Error** line and, instead of returning an acknowledgement from the requested DMA operation, returns an information packet marked as error information. (See Figure 4-5.) This same mechanism is used to report errors on the LSI-11 bus during a requested DMA operation (i.e., memory parity or non-existent memory). The error packet is returned to the Pmap, where the associated context is restarted with an error code.

3. Transfer of data to a processor, following a non-local read operation. This must be treated as a special case. Normally, if no errors are detected, the Pmap context associated with the transaction is released by the Kbus when the operation is successfully completed. Releasing the context is equivalent to destroying all information associated with the request. To ensure that data is correctly returned to the processor before releasing the context, this final data transfer is performed in a partially interlocked manner. If the Slocal receives bad parity on data intended for the processor, the data is ignored and the Map Bus **Flag** signal is asserted. This signal is inspected by the Kbus before releasing the context. If asserted, the Kbus can retry the transmission several times. If this does not lead to success, the context associated with the transaction is restarted with an error code. The context can also retry the transmission several times before attempting to send an error report to the processor. If some fault in the Slocal of the requesting processor is causing all data received to be considered in error, then there is no way to make it accept even an error report. The processor will eventually take an error trap, with an indication of Kmap timeout.

4.3.7 Why Individual Map Bus Request Lines?

We noted earlier in this section that there are a total of 28 individual service and return request lines from the Cm's to the Kbus. Even when effectively divided in two by partitioning the Map Bus, this is an unusually large number of wires for an arbitration scheme. Thus a little justification is in order.

The speed and cost advantages of central arbitration are obvious; Unibus style distributed arbitration (with a daisy chained grant) imposes a delay of 100 to 200 ns for each unit on the chain and costs a minimum of 3 IC's per unit[1]. For a cluster with 14 Cm's, this would add up to at least 82 IC's (two arbitration circuits per Cm) with an average arbitration delay of 1,000 to 2,000 ns. The Kbus performs arbitration in an average of 300 ns with a total of 28 IC's. Central arbitration also simplifies implementation of the required "Round-Robin" service

[1]This assumes using DEC's custom LSI Unibus arbitration chip plus the bare minimum of logic to provide a round-robin service discipline.

discipline.

Depending on the application domain, this substantial cost and performance advantage might not, in itself, compensate for the restriction of extensibility of the number of Cm's in the cluster and the cost of connector pins and cabling for the request lines--particularly if a re-implementation with LSI is contemplated. However, there are some other, less obvious, advantages to taking individual request lines to the Kbus. Principally, these relate to the fact that an individual request line identifies immediately the requesting unit and allows individual requests to be selectively enabled and disabled.

1. A Service Request, once it is allocated a Pmap context, is disabled so that it does not enter into arbitration. Should the transaction ever get aborted, for example because it is a reference to another cluster and is allocated to the reserved context (see Section 3.4), then it can be restarted (without any reference to the Cm) by simply enabling the request line.

2. Another way that the selective disabling and re-enabling of requests, discussed in (1) above, can be used is for a demand paging system. If a processor references a segment which is not in primary memory, then it can be suspended, with its request line (and timeout) disabled, indefinitely while the segment is brought into memory. An automatic mechanism, driven by the timeout mechanism within the Kmap, can periodically restart all suspended requests. A request suspended in this way does not consume any resources, other than the suspended processor. (Demand paging has not yet been implemented on Cm*.)

3. This mechanism can be used to suspend a processor when it attempts to access a segment which is being moved.

4. For Return Requests, the individual request line immediately identifies the requesting Cm. This is used to index a small table in the Kbus to derive the context number, restart status and original source processor number. For normal Return operations, the Kbus can immediately place the source processor number on the bus, as the Destination Cm, and data can be returned directly to the requesting processor.

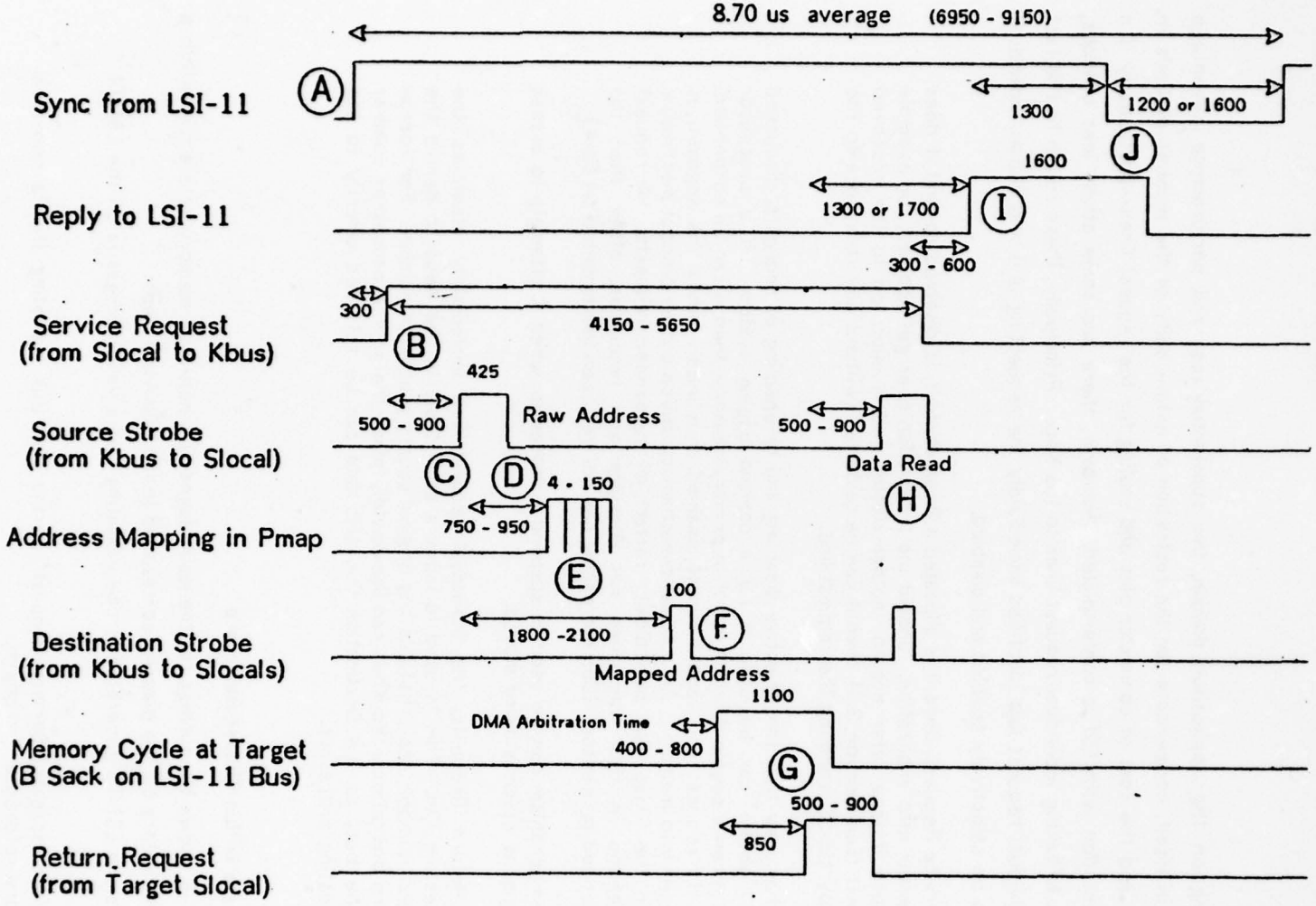
4.3.8 A Single Within Cluster Reference

Figure 4-6 shows the timings for the main stages of a non-local memory reference within a cluster. The letters on the diagram correspond to the following steps:

A. Sync is the LSI-11 generated strobe indicating that a valid address is on the LSI-11 bus.

B. The Slocal asserts "Service Request" on the Map Bus indicating that a non-local memory reference has begun.

C. The Kbus, after the "Service Request" has gone through the round-robin arbitration logic, places the requesting Cm's address (4 bits) on the Map Bus and asserts "Source Strobe". The Slocal responds by enabling the address from the processor,



Timing Diagram Mapped Reference to Self.

Figure 4-6: Non-local Reference within a Cluster

Detailed Timings: Single Processor executing L: 137 L
Mapped to Self, no other map bus activity
Note: DMA Arbitration Time is slower when mapped to another processor
Cm 4 in cluster 2, May 24 1977

which it has latched, onto the Map Bus.

D. The Kbus latches the address and control signals, and queues a request for the Pmap.

E. The context corresponding to this request is activated, and the Pmap translates the 16 bit address from the processor to a physical address (Cm* plus 18 bits) within the cluster. The timings shown are for a simple version of the microcode. The microcode supporting the capability based addressing structure requires three extra 150 ns Pmap cycles. The translated address is queued from the Pmap to the Kbus.

F. The 18 bit physical address is transmitted over the Map Bus to the target Cm. The Slocal latches the address and immediately requests a DMA cycle on the target LSI-11 bus.

G. The memory reference on the LSI-11 bus is performed. The Slocal latches the data read and asserts "Return Request" on the Map Bus.

H. After arbitration, the Kbus places the address of the target Cm (as source) and the address of the originally requesting Cm (as destination) on the Map Bus and asserts batch "Source" and "Destination" strobe. The Kbus determines the original requesting Cm by using the target Cm number to index a small, private table. This table within the Kbus also contains the number of the context handling the transaction. Because the transaction was completed without error, the context is now marked as available by the Kbus. If an error was reported by the target Slocal or if no response occurred within 2 ms, the context would be awakened with an error status.

I. At the source Cm, the Slocal passes the data read back to the processor.

J. The de-assertion of sync indicates the end of the memory cycle.

4.3.8.1 The Source of Delays for within Cluster References

The timing diagram, Figure 4-6, indicates that about 6 us have been added to the normal interference time of 3 to 3.5 us. (i.e., for all non-local references a program would run about three times slower). We can examine where this overhead was incurred, in order of decreasing affect:

- About 2.7 us (or 29% of the total of 9.3 us between successive memory references) is due to time required for the processor to consume one word and generate the address of the next. This is independent of whether the reference is local or non-local.

- About 1.5 us, or 16%, is due to the three Map Bus transactions at 0.5 us each. (The Kbus and Map Bus are occupied for a further 0.5 us during the last transaction because it is interlocked for error reporting.)

- About 1.3 us, or 14%, is lost to arbitration for the LSI-11 bus at the target Cm.

- Similarly, 1.3 us or 14% is lost to arbitration for the Kbus and Map Bus.

- About 0.85 us, or 9%, is required to perform the memory reference at the target Cm. (From acquisition of the LSI-11 bus until valid data is available and the "Return Request" to the Kbus is asserted.)

- 0.75 us, or 8%, is required for the address mapping by the Pmap. However an additional 0.3 us (or 3%) delay occurs in starting up the context.

From the point of view of an individual processor the above delays are all additive and all contributed to degraded performance. With respect to the performance of the cluster as a whole, only the Pmap and Kbus-Map Bus combination are critical, central resources. The Pmap is occupied for only 8% of the total period of a non-local reference. This may be overlapped with the use of the Map Bus by some other request. The Kbus-Map Bus combination is occupied for about 21% of a non-local reference. These figures, and those in the following section, show that the packet-switching implementation has been successful in improving bus utilization. With a circuit-switched implementation, the Map Bus would be occupied for 100% of a transaction.

4.4 Intercluster Bus Protocol

It is instructive to examine how different objectives and assumptions led to the design of a bus with a protocol very different from that of the Map Bus. It has been argued that the intercluster bus should be controlled by a Kmap. This would be consistent with a hierarchical view of the structure of Cm*. However, from a structural, although not necessarily a conceptual, viewpoint it is desirable to minimize the number of hierarchical levels. There are two motivations for this: reliability and performance. Often, if a component in a hierarchy fails, all components below that level must cease operation--or at least act in isolation. Again often, but by no means necessarily, the component at the highest level in a hierarchy can limit overall system performance.

The view taken for the structure of Cm* is that each cluster should be as independent and autonomous as possible. For communication between clusters it is necessary to rely on the interconnecting medium. The medium is a set of copper wires without any central control or arbitration. Being entirely passive, this should provide greater reliability than any scheme depending on central active components. In addition, the two intercluster buses per Kmap can be arranged to give an alternative access path in the event of bus failure.

In the discussion of the Map Bus design, Section 4.3.3, two important restrictions were noted which led to higher performance and lower cost. The number of Cm's per Map Bus is limited to 14, and they are constrained to be physically close. These restrictions are appropriate for a single cluster; they would be very burdensome for the interconnections

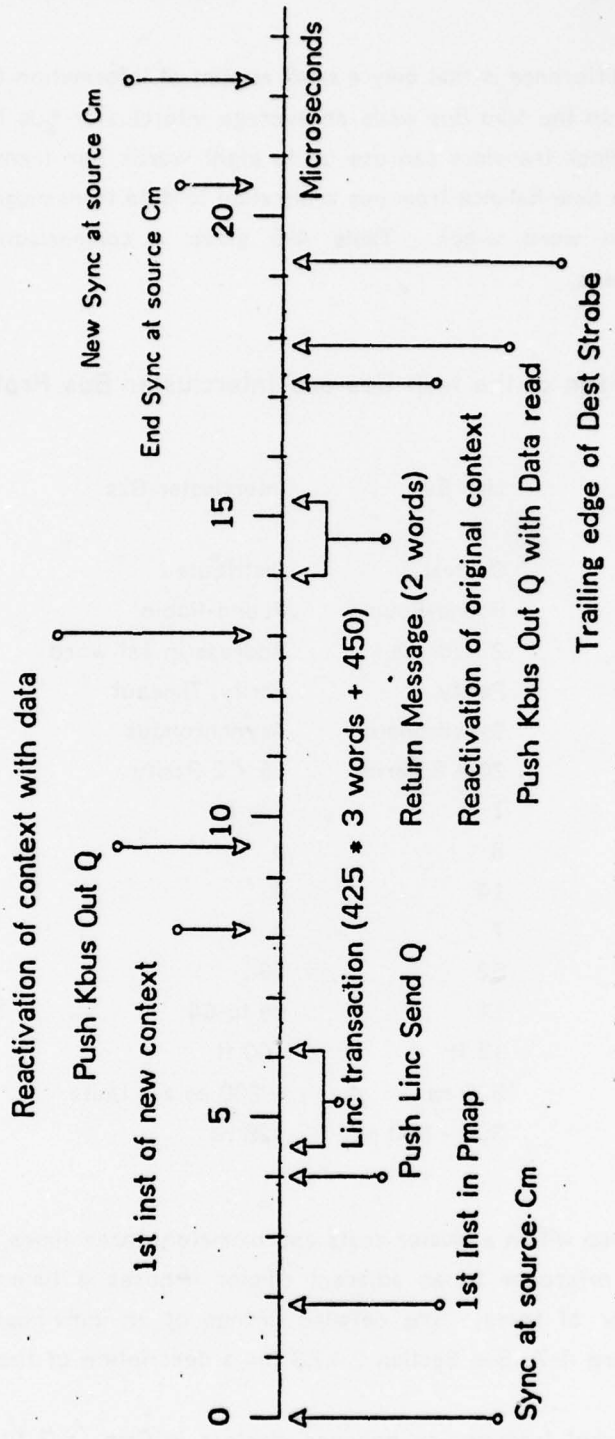
between clusters. Another difference is that only a small amount of information (one word) is transferred per transaction on the Map Bus while an average intercluster bus transaction is two and one half words. (Block transfers can use up to eight words per transaction.) This has the affect of shifting the time balance from bus arbitration to data transmission (assuming the data path is only one word wide). Table 4-5 gives a comparison of various characteristics of the two buses.

Table 4-5: Comparison of the Map Bus and Intercluster Bus Protocols

	Map Bus	Intercluster Bus
Arbitration	Central	Distributed
Priority Scheme	Round-Robin	Round-Robin
Unit Addressing	2 Address	Address in 1st word
Error Control	Parity	Parity, Timeout
Data Transfers	Synchronous	Asynchronous
Data Path Width	20 + 3 Parity	16 + 2 Parity
Transfers/Transaction	1	1 → 8
Addressing Signals	8	0
Arbitration Signals	14	4
Other Control Signals	7	4
Total Signals	52	26
Max # Units/Bus	14	up to 64
Max Bus Length	15 ft	100 ft
Arbitration Latency	300 ns	~ 200 ns * # Units
Transfer Time/Word	300 - 500 ns	425 ns

Where a non-local reference within a cluster costs approximately three times as much as a local memory reference, a reference to an adjacent cluster imposes a time overhead of approximately another factor of three. The detailed timings of an individual intercluster reference are shown in Figure 4-7. (See Section 2.4.2.3 for a description of the progress of an intercluster reference.)

The response time for packet transmission between clusters in Cm*, is 3 to 5 order's of magnitude faster than that in the ARPAnet. This is due, in part, to closer physical spacing and



Inter-Cluster Reference: Reference mapped back to same Cm.

Average Period between references: 22.4 us. (2.9 us all local)
Kmap 1, May 26, 1977

Figure 4-7: The Timing of an Intercluster Reference

higher bandwidth communication links. The delays associated with allocating, de-allocating and transmitting packet-buffers have been made minimal through implementation in the Linc hardware. (See Section 2.4.1.) This hardware implementation is greatly facilitated by the small upper-bound (8 words) on packet size. A major factor in the response time in the ARPAnet is delays in the Host operating system. In Cm* this corresponds to the delay incurred in allocating a context in the destination Kmap and the time to perform the requested memory reference (less than 10 us.).

4.5 Using Commercial Uniprocessors in a Multiprocessor

Ideally, the problems of adapting a commercially built processor for use in a multiprocessor would never arise. In an ideal world, an entirely new processor would be designed which would be exactly tailored to the needs of the switching structure, architecture and operating system of the new multiprocessor structure. History indicates that we do not live in an ideal world; Pluribus, C.mmp and Cm* all faced the problems of modifying inadequate processors. In no case was the decision to use a commercial processor, rather than designing a new one, entered into lightly. There are many strong technical and commercial reasons favoring use of an established commercial processor:

1. **Software.** Commercial processors have available an array of compilers, loaders, diagnostics, debuggers, peripheral handlers, etc. All this software would have to be duplicated if a new processor, with a new instruction set, was created.

2. **Hardware.** The design and implementation of a new processor is in itself a major engineering task. In addition, a new computer design probably requires implementation of new interfaces for standard peripherals, for example, disks, terminals, etc.

3. **Familiarity.** When designing an experimental computer system, there is much to be said for keeping as much as possible of the system familiar to potential builders and users.

4. **Cost.** Mass production of mini and microcomputers leads to excellent cost-performance relative to larger computers. Particularly for microcomputers, it is essential to have a large market to cover development costs. It is the promise of using inexpensive, mass produced microprocessors to build large computer systems that makes multiprocessors so attractive. Thus it might be argued that the cost-performance advantage of multiprocessors depends on them utilizing processors intended for uniprocessor use.

Even taken collectively, these arguments are not completely convincing. The amount of software which can be salvaged from other systems may be small relative to the new software which must be written anyway. The Hydra project on C.mmp made extensive use of Bliss-11 compilers--but to some extent the compiler was written for this purpose. It did not

use any other pre-existing software. The Cm* project has used a substantial amount of software originally developed for C.mmp: This included Bliss-11, the Algol 68 system, the Harpy speech understanding system, and several other benchmark applications. The only software salvaged from uniprocessor use were the hardware diagnostics.

The issue of interfaces for standard peripherals might be met by using a standard bus structure or providing an adaptor. C.mmp uses standard interfaces for disks, line printers and terminals. Cm* uses standard memory and terminal interfaces; other interfaces were specially designed for Cm*.

On the question of cost, experience with C.mmp and Cm* suggests that it may be cheaper to build a new processor than modify an existing design. With C.mmp, about 150 IC's can be attributed to modifying and extending the power of each PDP-11/40 (itself about 400 IC's). With Cm*, we will see that more than 80 IC's are used to modify and extend the power of a basic LSI-11 (itself consisting of about 80 16-pin-equivalent IC's). (This comparison does not include a further 60 IC's attributable to the Map Bus interface and DMA access on the LSI-11 bus. See Table 4-6.) For both C.mmp and Cm* it would require a detailed design effort to determine the cost of constructing processors of similar performance, using comparable technology, designed specifically for the task.

Nevertheless, Cm* was constructed using modified commercial processors. Apart from the considerations given above, a major motivation was a strong desire to get a working system up quickly--to concentrate on the research issues peculiar to multiprocessors without diverting energy into processor design. An important, but non-technical consideration was that large amounts of commercial equipment were available free.

4.6 Facilities Provided by the Slocal

The Slocal (PMS notation for local Switch) was introduced in Chapter 2. Its role in the implementation of packet switching between the Map Bus and LSI-11 bus was discussed in Sections 3.3.1 and 4.3. In this, and succeeding, sections we will examine how it is used to modify and extend the LSI-11 processors and provide numerous other facilities.

4.6.1 Physical Placement of the Slocal

The Slocal is a single large (160 IC) wirewrap board mounted adjacent to the LSI-11 processor in each computer module. It has connections to the LSI-11 bus, the Map Bus and direct connections to signals on the processor board. A number of connections on the processor (printed circuit) board have been cut, wires run to logic on the Slocal, and signals

Table 4-6: Chip Usage in the Slocal

Extending the power of the LSI-11 Processor	
Relocation of Local Memory References	10
Privileged Instruction Detection	4
Last Fetch Address, Fetch Indicator and Reference Count	7
Local Error Reg, Initialization, etc.	10
External PSW, Trap Sequence Detection	9
Multilevel Interrupt Scheme, Local Interrupts	6
Access to Slocal Registers	17
Clock Drivers, Misc.	2
	65
Forcing Pc off bus during mapped reference, etc.	17
Map Bus - LSI-11 Bus Interface (Additional to logic necessary for above)	
Map Bus Control Logic	14
Map Bus Drivers, Parity	14
Inter Bus Transfer Registers, etc	12
Map Bus Terminators (Usually not used)	4
LSI-11 Bus DMA Interface (B Dal Drivers counted above)	18
	62
Grand total	142

All counts are approximate only, most chips are 14 or 16 pin MSI.

returned to the processor board[1]. This method was necessary for the implementation of relocation and forcing the processor off the bus during non-local memory references.

4.6.2 Modifying the LSI-11 for Packet Switching

The decision to use a commercially available processor, see Section 4.5, has had a substantial affect on the structure of a Cm. Figure 4-8 shows a simplified view of the structure of a Cm. Unlike the structure shown in Figure 3-5, there is no separation of "active" and "passive" devices. The local memory bus is allocated, to the processor, for both local and non-local memory references. Local references are performed in the same way as with an unmodified LSI-11, except for relocation and physical address expansion . When the Slocal detects, on the basis of the address generated, that a reference is non-local it takes special action: The LSI-11 generated address (and in the case of a write, data) is latched for access via the Map Bus and service from the Kmap is requested. To avoid a deadlock potential, it is now necessary to cause the processor to relinquish the memory bus--although the reference is not complete. There are two cases:

1. *Read Operations.* The bus control signals asserted during a read operation by the processor (SYNC and DIN) are gated by logic on the Slocal. Thus these signals are forcibly de-asserted. The bus arbitration logic on the processor board believes that the bus is in use via the processor and so will not respond to DMA (Direct Memory Access) requests from devices (including the Slocal) on the memory bus. The Slocal takes over the role of bus arbitration for the duration of the non-local read operation. Thus the memory bus is fully available for accesses by remote processors (also performed by the Slocal) and local DMA devices.

   When the remote read is complete, the Slocal waits until the memory bus is free and places the returned data on it. It then asserts the acknowledgement signal (Reply) to the processor to inform it that data is available and the processor completes the reference in the normal way. It is necessary for the Slocal to block the processor from receiving any Reply signal from DMA activity while the processor is suspended.

2. *Write Operations.* A write cycle cannot be suspended in the middle as described for a read operation above. This is because the processor continues to assert data on the LSI-11 bus until given an acknowledgement of the memory cycle completion. It would be possible to disable the bus drivers on the processor to free the bus; however the 4K words on the processor board would be inaccessible because they share bus drivers (and some on-board data paths) with the processor. (Processors in Cm*/50 do not have on-board memory.)

   Rather than suspend the processor in mid cycle, the Slocal latches the data to be written and asserts Reply. The processor completes the bus operation and releases the bus. The Slocal then suspends further processor activity (by

[1] A total of 35 signals.

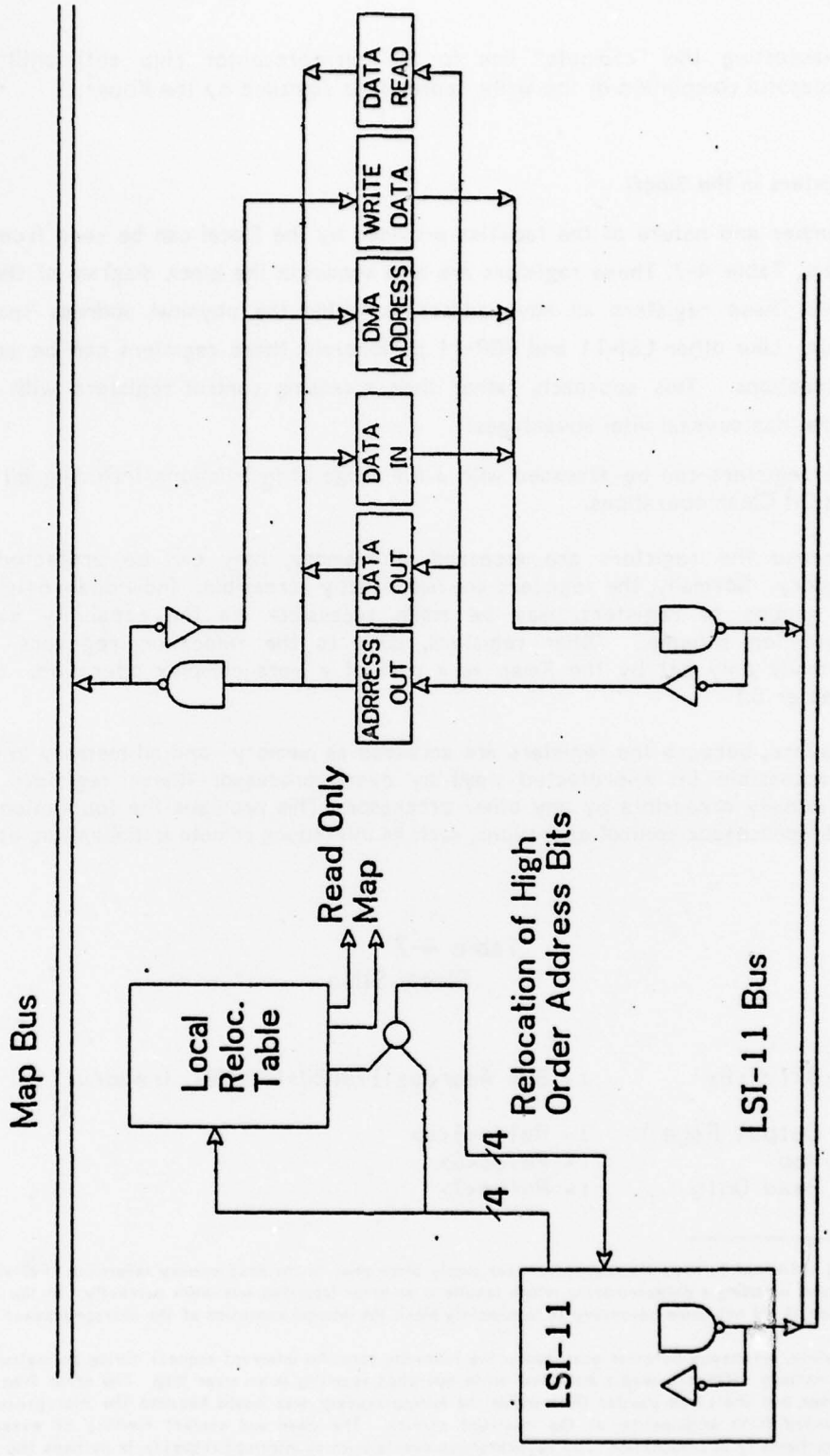


Figure 4-8: Simplified LSI-11 -- Slocal Data Paths

de-asserting the "compute" line to the microprocessor chip set) until the successful completion of the write operation is signalled by the Kbus[1].

4.6.3 Registers in the Slocal

The number and nature of the facilities provided by the Slocal can be seen from the table of registers, Table 4-7. These registers are also shown in the block diagram of the Slocal in Figure 4-9. These registers all have addresses within the physical address space of the LSI-11 bus. Like other LSI-11 and PDP-11 peripherals, these registers can be accessed as memory locations. This approach, rather than accessing control registers with special IO instructions, has several vital advantages:

1. The registers can be accessed with a full range of instructions, including Bit Set and Bit Clear operations.

2. Because the registers are accessed as memory, they can be protected as memory. Normally, the registers are not directly accessible. Individual registers, or groups of registers, may be made accessible via the capability based protection scheme. Other registers, such as the relocation registers, are normally only set by the Kmap as a part of a more complex operation. (See Chapter 6.)

3. Likewise, because the registers are accessed as memory--and all memory in Cm* is accessible (in a protected way) by every processor--these registers are potentially accessible by any other processor. This provides the foundation for inter-processor control operations, such as interrupts, remote initialization, etc.

Table 4-7
Slocal State

```
RELOC[0:37]<7:0>          := Bus Address[170000:170076]  (read/write)

    Output Page           := Reloc<5:0>
    Map                   := Reloc<6>
    Read Only             := Reloc<7>
```

---

[1] It is not sufficient to have the microprocessor simply block prior to the next memory reference. For almost every situation, including a detected error which results in an error trap, this will work correctly. In the initial implementation, it did not seem necessary to immediately block the internal execution of the microprocessor.

Unfortunately, erroneous behavior occurred in the following case: An interrupt request during an instruction whose last memory reference was a non-local write operation resulting in an error trap. The error trap was correctly taken but the stack pointer (R6) within the microprocessor was invalid because the microprocessor had decremented it in anticipation of the interrupt service. The clean and correct handling of exception conditions, particularly in conjunction with asynchronous events such as interrupt requests, is perhaps the most difficult aspect of processor design.

No automatic initialization is provided.

```
XPSW<15:8>              := Bus Address[170100]<15:8>

      Space H           := XPSW<15>     ; Selects user or Kernel Address
                                          space
      God L             := XPSW<14>     ; Status bit interpreted by the
                                          Kmap
      Privileged Instr  := XPSW<13>     ; When '0', enables the execution
                                          of HALT & RESET  etc.
      Control Stack H   := XPSW<12>     ; Enables protected Control
                                          Stack & Interrupt Vector ops
      Reloc Mode H      := XPSW<11>     ; Enables Relocation Table
      Interrupt Enb<2>  := XPSW<10>     ; Enables Interrupt requests
                                          from BIRQ 2 L to be sent to
                                          the processor.
      Interrupt Enb<1>  := XPSW<9>      ; Enables Interrupt Requests
                                          from BIRQ 1 L to be sent to
                                          the processor.
                                          This also enables events
                                          on the BEVNT L to be processed
                                          When '1', writes to Interrupt
                                          Req[1] will cause interrupts
                                          to be generated.
      Interrupt Enb<0>  := XPSW<8>      ; When '1', writes to Interrupt
                                          Req[0] will cause interrupts to
                                          be generated.
   All bits initialized to '0'. Slocal clear sets all bits to '0'.

Local Error<7:0>        := Bus Address[170102]  (read only)

      Parity Error      :=Local Error<7>; Parity Error on a processor
                                          reference to local memory.
      Forced NXM        :=Local Error<6>; Error trap was forced by
                                          write to Force NXM bit.
      Privileged Instr  :=Local Error<5>; Attempt to execute privileged
                                          instruction when XPSW<13> = 1.
      Kmap Timeout      :=Local Error<4>; Kmap did not respond within
                                          250ms to a request.
      Mapped            :=Local Error<3>; The reference that caused the
                                          error was mapped.
      Ref Count<2:0>    :=Local Error<2:0>
                                        ; Memory references from the
                                          last instruction fetch
                                          excluding the instruction.
   A write to this address will clear bits <7:4>.

Force NXM               := Bus address[170104]<0>
                                        ; A '1' written into bit<0> will
                                          force a NXM trap.

Processor Clear         := Bus Address[170104]<1>
```

```
                                        ; A '1' written into bit<1> will
                                          send a DCOK to the processor.

Slocal Clear            := Bus Address[170104]<2>
                                        ; A '1' written into bit<2> will
                                          send a clear to the Slocal.

Fetch Address           := Bus Address[170106] (Read only)
                                        ; Address of the instruction
                                          fetched prior to a local error
                                          This is the address generated
                                          by the processor, and not the
                                          relocated address

Interrupt Req[0:1]      := Bus Address[170110:112]<7> (r/w init 0)
                                        ; Writing a '1' into bit<7>
                                          will interrupt the processor
                                          at vectors #40 & #44 resply.
                                          If bit<7> is set when read
                                          it indicates an interrupt
                                          request is pending.

Parity CSR              := Bus Address[170114]

       Write Wrong      := Parity CSR<15>
                                        ; Parity written when this is
            Parity                        set will be different from
                                          normal parity.
        Error Address<17:16> := Parity CSR<14:13>
                                        ; Higher order two
                                          bits of the error address
        Dma Reference    := Parity Csr<10>
                                        ; Error occurred during a Dma
                                          cycle
        External Ref     := Parity Csr<9>
                                        ; Error occurred during an
                                          reference from an other Cm.
        Clear            := Parity Csr<8>
                                        ; A '1' written to this bit will
                                          clear the error bits <14:13>
                                          <10:9,7,5:4,0>.
        Parity Error     := Parity Csr<7>
                                        ; Indicates a parity  error
                                          occurred.
        Interrupt Enb    := Parity Csr<6>
                                        ; If one, a parity error will
                                          cause an interrupt.
        NXM Error        := Parity Csr<5>
                                        ; Indicates a NXM error occurred
        Rep Parity Er    := Parity Csr<4>
                                        ; Indicates more than one parity
                                          error occurred since the CSR
                                          was last cleared.
```

```
        Parity Check Enb:= Parity Csr<0>
                                        ; If one, a Parity Error will
                                          force a NXM to the processor.
Error Address<15:0>     := Bus Address[170116]
                                        ; Address of reference that
                                          caused either a parity or a
                                          NXM error.
Disable Kmap Timeout    := Bus Address[170120]<7>(r/w init 0)
                                        ; Processor will wait for the
                                          Kmap indefinitely.
Enable Line Clock       := Bus Address[170122]<7>(r/w init 0)

Hold LSI-11 Bus         := Bus Address[170124]<7>(r/w init 0)
                                        ; Keeps Processor off bus and
                                          reduces Dma latency
Halt Processor          := Bus Address[170126]<7>(r/w init 0)
                                        ; Asserts LSI-11 halt line.
Not Used                := Bus Address[170130:170134]

Run                     := Bus Address[170136]<7>(read only)
                                        ; Indicates the Processor
                                          fetched an instruction
                                          within the last 250ms.
Not Used                := Bus Address[170140:170176]
                                        ; Respond to these addresses
```

4.6.4 LSI-11 Compatibility and the X PSW

The LSI-11 processor implements only the low order byte of the standard PDP-11 Processor Status Word (PSW). The PSW contains conditions codes, interrupt enable and other status information which is saved and restored automatically during interrupt and trap operations. It can also be read and set explicitly under program control. To provide further control and status bits, the Slocal implements the high order byte of the PSW. This is called the X PSW<15:8> (eXternal Processor Status Word). The X PSW is loaded and saved in conjunction with the internal PSW. However, the instruction for explicitly setting the internal PSW does not affect the X PSW.

Several of the control bits within the X PSW selectively enable various facilities provided by the Slocal, see Table 4-7. It is important, for the sake of initialization, testing and protection, to have individual program control of these facilities.

In the initialized state, after power up or an externally forced reset, the computer module is compatible with standard LSI-11 diagnostic programs and has no access to the rest of the Cm* system. However, the memory and registers on the LSI-11 bus are accessible from elsewhere in the Cm* structure--thus bootstrap and test programs can be loaded.

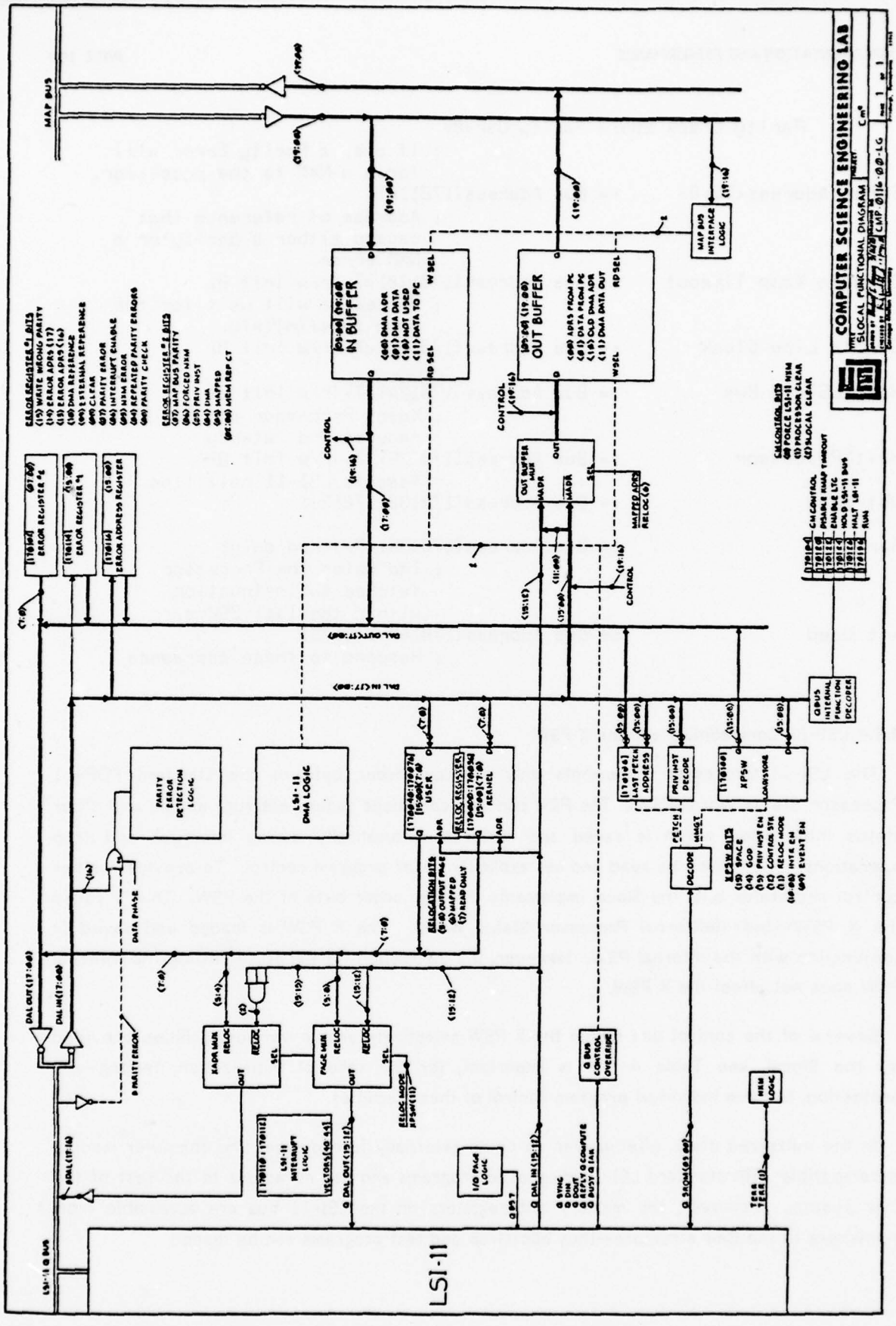


Figure 4-9: A Block Diagram of the Slocal

Setting X PSW<11>, Relocation Mode, enables the relocation of local memory references and the passing of references to the Kmap. There are two sets of relocation registers, Reloc[0:15] and Reloc[16:31], providing two independent address spaces. The current address space, user or kernel, is selected by the Space bit in the X PSW. This allows for an automatic change of address space when servicing an interrupt or trap[1]. See Section 2.4.2.1 for a description of the relocation mechanism.

Other control bits in the X PSW include Privileged Instruction Enable and Control Stack Enable. See Section 4.6.7. The control bits Interrupt Enable<2:0> are part of a simple multilevel interrupt masking scheme. The standard LSI-11 processor provides only a single interrupt level; this has been expanded to three levels.

4.6.5 Interprocessor Control Operations

The philosophy of Cm* is that all operations be represented as memory references. A good example of this is low level interprocessor control operations. These are invoked by write operations to various registers in the Slocal. There are two, independent interrupt request flags which invoke conventional vectored interrupts. The requests may be individually masked by interrupt enable bits in the X PSW. A Cm can also be halted and initialized by setting write operations to the appropriate control registers.

If during a non-local memory reference, the Kmap detects an unrecoverable error (for example, invalid memory parity), it is necessary to report the error to the requesting processor. The Kmap does this via the Force NXM bit in the Slocal. This unblocks the processor from waiting for acknowledgement of a non-local reference, and forces an error trap. In most computer systems, this very low level control operation would be achieved by a special control line on the memory bus or by passing back a special error code. Apart from important conceptual advantages, which may be a question of taste, this has the practical advantage that the error trap mechanism can be directly tested under program control (and without aid from the Kmap).

4.6.6 Error Recovery Information

A frequent problem for users and maintainers of a computer system is that the hardware provides little or no information about the circumstances surrounding a detected error. For example, in most members of the PDP-11 family including the LSI-11, after a memory bus

---

[1] Use of one or more bits in the PSW to select the current address space is standard for many members of the PDP-11 family.

error it is not usually possible to determine the address of the instruction being executed when the error occurred, let alone recover and retry the instruction[1]. Throughout the design of Cm* an effort has been made to capture, and make available to the programmer, as much error information as possible. Table 4-8 indicates many of the classes of errors detected in hardware.

For non-local references, much of this information is captured by the Kmap and made available via pseudo registers in the address space of the process (see Section 6.6.1). For both local and non-local references the Slocal captures some low level status information. All error conditions result in the setting and freezing of the Local Error register. (See Table 4-7.) This gives the source of the error (parity on local memory reference, attempt to execute a privileged instruction, error trap via Force NXM bit, or Kmap timeout) and some key status information. The Error Register indicates whether the error occurred during a local or mapped reference and the number of memory references since the last instruction fetch. The Fetch Address register captures the address of the first word of the last instruction fetch. Knowledge of the instruction address and progress through the instruction (the number of memory references) uniquely determines the state reached by the processor. This does not mean that it is necessarily possible to restore the state of the processor and re-execute the instruction. For example, John Ousterhout points out that operands on the stack may have been overwritten if multiword arithmetic operations are aborted in mid-execution.

The LSI-11 bus memory provides parity generation and checking. The control and status registers for the parity logic are provided on the Slocal. If a parity error is detected, the address of the erroneous word and the source of the reference is saved. There are three sources distinguished:

1. A parity error during a local memory reference causes the processor to take an error trap. (For uniformity, a parity error interrupt is also requested.)

2. A parity error detected during a memory reference performed by the Slocal, for some remote processor, causes an error indication to be passed back to the Kmap and a parity error interrupt to be requested. It is the local processor's responsibility to reset the parity control logic.

3. Parity is also checked for references by local DMA devices, such as disks. Many standard peripherals do not check for parity errors; those designed at CMU for Cm* do. The Slocal will always request a parity error interrupt.

---

[1]The error trap routine has access to the previous Program Counter; however it has no way of determining how far through a multiword instruction execution had progressed. Thus it cannot determine, in all cases, the address of the first word of a multiword instruction. Even if the instruction address can be determined, side affects on processor registers during instruction execution, make recovery very difficult.

Table 4-8: Error Detection and Reporting

Local Memory References

Error	Detected by:	Reported to:
LSI-11 Memory Parity	Slocal and memory	Processor
LSI-11 Bus, No Response	Processor	Processor
Attempt to write read-only segment	Slocal	Kmap
Attempt to execute privileged instruction.	Slocal	Processor

Non-Local Memory Reference

Error	Detected by:	Reported to:
LSI-11 Memory Parity with memory	Slocal and Memory	Kmap and processor with memory
LSI-11 Bus, No Response	Slocal	Kmap
Slocal, No Response	Kmap	source processor
Map Bus Parity	Slocals and Kbus	Kmap
Pmap Control store and Data RAM Parity	Pmap	"Hooks" processor
Kmap, No Response	Slocal	processor

4.6.7 Protection Facilities

The LSI-11 is an inexpensive microcomputer that was not intended for use in an operating system environment. Consequently it has no facilities for protecting the system against erroneous or malicious behavior by a program. The Slocal provides the basic hardware necessary for implementing protection. The capability based protection scheme provided by the Cm* architecture is built on top of this, using Kmap microcode. (See Chapter 6.)

As already noted, the Slocal provides two independent address spaces so that the kernel can execute within a separate address space from user programs. Within an address space, access rights to segments are determined by capabilities. For non-local accesses this protection is checked by the Kmap. However, it is also necessary to have protection for access to segments referenced locally. Only 2K word segments in local memory can be directly accessed. Read-only access can be provided by setting the **Read Only** bit in the Reloc register for that page. Although read references will be performed locally, any write referenced will be automatically passed to the Kmap. (Just as if the "map" bit were set.) This permits "rights" violations on local references to be treated identically to violations on non-local references. This mechanism is also used to ensure that a "dirty" bit is set on writes to local segments.

There are certain instructions which can cause control of the processor to be lost (Halt) or enable bypassing of memory protection (Reset, various Trap instructions, Return from Interrupt, etc.). To prevent this, the Slocal has a table (in read only memory) of **Privileged Instructions**. Unless **Privileged Instruction Enable** in the X PSW is set, an attempt to execute any of the designated instructions will cause an error trap.

The Slocal provides a further form of protection which involves a close relationship between the Slocal hardware and the addressing architecture of Cm*. This is discussed in the following section.

4.7 The PDP-11 Stack Protection Problem

In the more primitive members of the PDP-11 family, notably PDP-11/20's, PDP-11/40's (without memory management) and LSI-11's, there is a single stack pointer. This is used by user programs and is directly accessible as Register 6. It must also be used by the operating system and for the automatic saving of processor state when an interrupt is serviced. In any system where the integrity of the overall system must be isolated from the actions of individual user programs, it is essential that interrupts be correctly serviced at any time. In an unmodified processor, if the stack pointer points to a non-existent, or write-protected, part of memory an attempt to respond to an interrupt will cause:

1. A non-existent memory trap.

2. In responding to the trap, the processor again attempts to use the stack to save state.

3. A "double bus error" is reported; on an LSI-11 this causes control to revert to ODT (a microcoded debugging system) which is equivalent to a Halt.

4. Even if the processor could be automatically restarted, the source of the interrupt and the state of the offending user program would be lost.

On C.mmp this unacceptable situation is rectified by a hardware modification:

1. The high order three bits of the stack pointer are made permanently 0. This constrains the stack to always be in the bottom page of (virtual) memory.

2. A hardware stack base register is added to protect the base region of the stack. An attempt by a user program to access this region will cause an error trap. This protected region includes an "overflow" stack section plus information about the process maintained by the operating system.

These hardware mechanisms ensure that an interrupt cannot be lost. The user cannot cause the stack pointer to point to any page other than the stack page (page 0) and if the user consumes the entire stack or sets the stack pointer to 0 (the last word in the downward growing stack) then an interrupt will cause the stack pointer to "wrap around" and use the overflow stack area. Each time the operating system is entered it is necessary to check, in software, that the stack pointer lies within specified bounds.

4.7.1 The Protected Control Stack on Cm*

The C.mmp solution, which is far from ideal, would be close to impossible to implement with an LSI-11. The basic processor is implemented with five LSI IC's. The necessary modifications would require changes to these IC's--this was not feasible.

The solution adopted for Cm* depends on the generalized notion of a segment supported by the addressing architecture. In conventional computers, a page or segment is simply a linear vector of words of memory. In Cm*, a segment is a typed object. A reference to a segment can invoke any one of eight operations defined for that object type. All segment operations are performed by microcode in the Kmap, except for simple reads and writes to 2K word segments which can be done locally by a processor. (See Chapter 6 for a full description of this.)

The protected control stack mechanism in Cm* utilizes a specially defined segment type,

called a **Stack Segment**[1]. A Stack Segment has an internal top of stack pointer. A "write" to a Stack Segment invokes a Push (the stack pointer is incremented and the data value is written onto the top of the stack). A "read" invokes a Pop (the data value at the top of the stack is returned and the stack pointer is decremented). Storage for a control stack is allocated in the normal way from primary memory. Approximately 28 microinstructions in the Kmap are required for implementation [Ousterhout, 77].

The address portion of a reference, for both read and write, is not used when accessing a Stack Segment. This is the key to making the operating system immune to user programs which invalidly set the processor's internal stack pointer (R6). We are concerned with the stack references which occur automatically during the response to an interrupt or trap. To ensure that an interrupt isn't lost, independent of the processor's own stack pointer, we must direct these references to the special Control Stack Segment.

Appropriate routing of references to a Stack Segment can be accomplished by keying-off a coded signal generated by the LSI, called MMGT (for Memory Management). This signal was provided by DEC to allow customers to detect changes to the Processor Status Word and implement some form of simple address expansion hardware. Figure 4-10 shows the sequences of memory references which can occur in conjunction with an interrupt, error traps, Return from Interrupt Instructions, etc. It is necessary to count up to four memory references following detection of MMGT. The first two references are directed to the control stack Segment[2]. In the case of an interrupt or trap service, there are two subsequent references to fetch a new Program Counter and Processor Status Word from the interrupt vector area. Again, to give protection to the operating system, the interrupt vector references are directed to a segment not accessible by user programs, rather than interpreting the addresses within the user's address space.

In summary, this mechanism gives full protection to the operating system for the service of interrupts and software traps. It utilizes a control Stack Segment and a segment for interrupt vectors which is accessible only to the operating system. The cost of the mechanism is about 4 IC's in the Slocal associated with each processor and 28 microinstructions in the Kmap. This entire mechanism would not be necessary on a processor designed to support an operating system.

---

[1] The control stack mechanism is presented here as a special case of the general segment notion in the architecture. In fact, the typed segment mechanism was developed as a generalization of a solution to this, and related problems [Swan, 75].

[2] Direction of the reference is accomplished by setting the four high order address bits to an otherwise inaccessible page (15) and forcing the reference to be passed to the Kmap. The Kmap treats the reference like any other non-local memory reference--it is not aware that the reference is part of an interrupt sequence. The operating system must ensure that a Stack Segment is bound to that page. See Chapter 6.

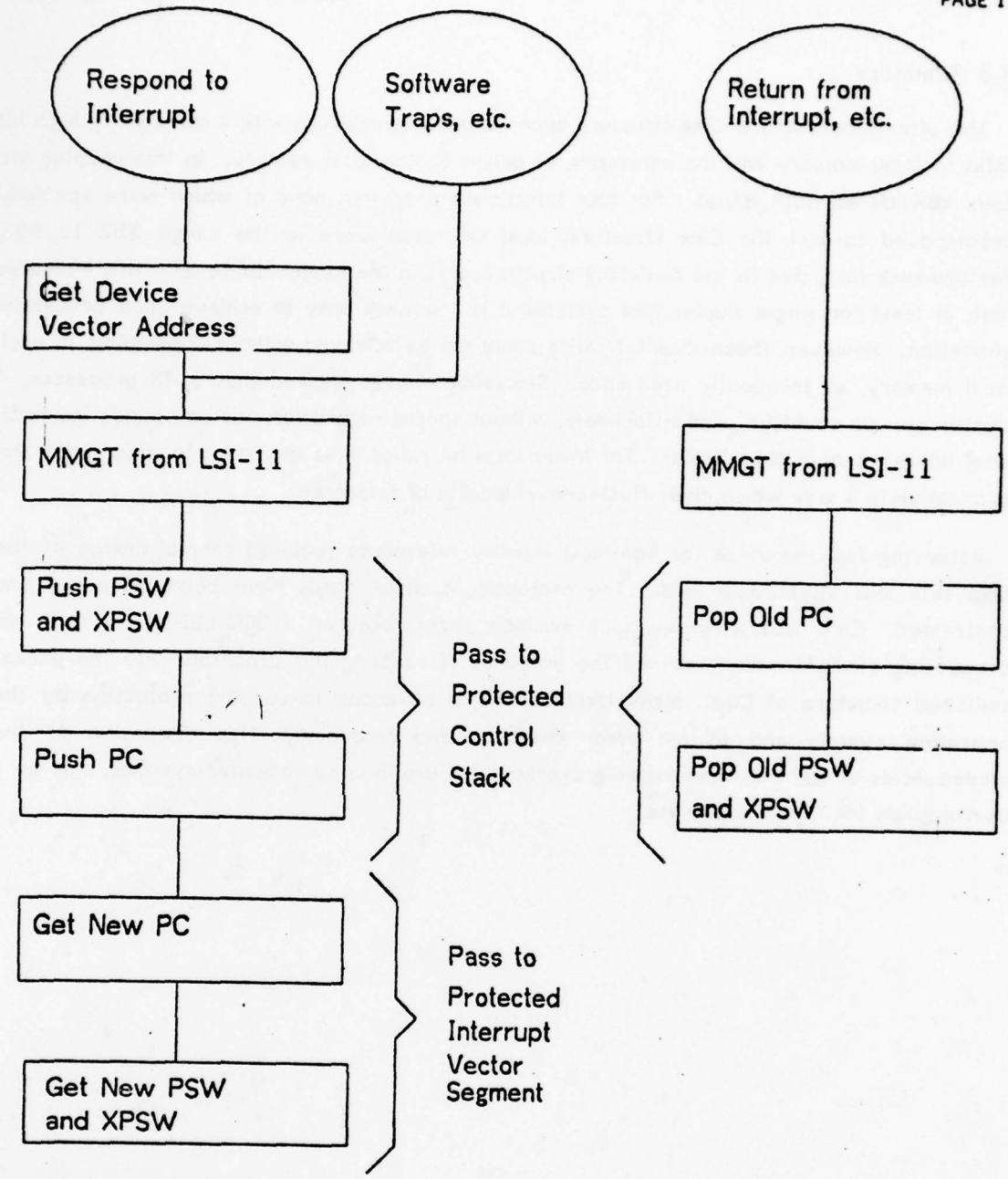


Figure 4-10: References to the Protected Control Stack and Interrupt Vector

4.8 Summary

The effectivness of the Cm* structure depends upon applications with a sufficiently high hit ratio to local memory and the minimizing of delays to non-local memory. In this chapter we have considered both issues. For four benchmark programs, none of which were specially decomposed to suit the Cm* structure, local hit ratios were in the range 85% to 99%. Performance loss, due to the switching structure, was in the range 22% to 2%. This indicates that, at least for single cluster Cm* systems, it is relatively easy to achieve good processor utilization. However, these local hit ratios could not be achieved without duplicating, in each local memory, all frequently used code. Simulation studies showed that a 48 processor, 4 cluster system could be used effectively, without special regard for cluster boundaries, with local hit ratios of 97% or better. For lower local hit ratios, it is important to decompose the application in a way which gives cluster-level locality of reference.

Achieving fast response for non-local memory references required careful design of the Map Bus and Intercluster Bus. The protocols of these buses have been discussed and contrasted. Cm* uses a commercially available microprocessor, a DEC LSI-11, as its basic processing element. We examined the problems of adapting this processor into the packet switched structure of Cm*. Many facilities had to be added to support protection by the operating system and to aid error detection and recovery. This discussion of the inadequacies of current inexpensive processors, for use in large computer systems, may be a useful guide for future designers.

5. A Model and Survey of Addressing Architecture

"Well here again that dont apply
But I've gotta use words when I talk to you"
Sweeney Agonistes - T. S. Elliot
-as popularized by J. G. Laski

5.1 The Importance of Addressing Architecture

The basic arithmetic, logical and control operations provided by conventional processors are remarkably similar. The major source of variation between instruction sets lies in the way operands are referenced. The method of specifying an access path to operands, including instructions, can be considered at two levels:

1. **Addressing Modes.** For example, direct register access, indirect through a register, auto-increment and indirect, top of stack, stack base plus displacement, etc., etc. Differences at this low level have their major impact on compilers and determine the suitability of the machine for particular languages.

2. **Object Reference.** At this level we are concerned with structure of the address space seen by a programmer and how these addresses are bound to physical memory locations. Addressing structure design, at this level, has a major effect on the operating system.

Neither of these levels of addressing structure design is well understood. There is no consensus about stack versus register architectures nor does there exist an accepted basis for comparing and analyzing designs within these major divisions. Similarly, at the object reference level there have been few attempts to evaluate specific designs or even taxonomize the design space.

A full investigation of this area will not be attempted here. However, this chapter will present a conceptual framework for the categorization, discussion and comparison of conventional mapping structures at the object reference level. Several brief examples, drawn from well known uniprocessor systems will be used to illustrate the application of the model. The practical implementation, during program execution, of address mapping structures will then be considered in the light of the model. It is only at this point that the particular relevance of this work to multiprocessors will become apparent. It will be shown that the conventional implementation of uniprocessor addressing structures on a multiprocessor leads to a significant functional loss. We will present alternative implementations for addressing structures and discuss their consequences for the design of multiprocessor switching structures.

5.2 A Model of Addressing Structures

In this section we will develop a simple framework for the representation of the address mapping structure normally interposed between a process and the physical memory of the processor executing that process. This model differs from the conventional presentation of the material and will attempt to emphasize the programmer's view of the mechanism. Denning, in his important expository paper on virtual memory [Denning, 70], gives a model of the implementation of address mapping and identifies some important desirable properties of addressing schemes. He makes the important point that that there exists a distinction between the 'Address Space' of a process and the 'Memory Space' of primary memory. Addressing schemes are then described in terms of the nature of the mapping between these spaces for an executing process. Recent operating system textbooks [Habermann, 76; Coffmann and Denning, 73] follow this model closely. Habermann [Habermann, 76] also points out that a translation between 'names' used in programs and 'addresses' used by a process is also provided. (This is normally performed by a language system and is beyond the scope of this thesis.)

Denning's model is adequate for investigation of the performance of multilevel memory management strategies. However, it is of limited use in the comparison and evaluation of addressing structures. Denning's model identifies two address spaces; we need three. In particular, his model does not acknowledge the existence of an 'object'[1] independent of the address used by a process to reference it or the physical locations used to hold it. Because of this, it does not allow representation of an object shared by two processes (except at the level of showing shared access to the same primary memory locations). Denning's two address space model is a good abstraction of the hardware and software used to implement address mapping in simple systems. It is not a good abstraction of a programmer's view of the overall addressing structure.

Abstractly, a computation can be represented as a sequence of operations on objects, or "information sets" [Lauer, 72]. These objects are given their significance (or meaning) by the programmer. The results of a computation depend on the value of the objects and the operations performed. The results do not depend on the particular integers ("virtual addresses") generated by the processor to reference the objects nor on the addresses of cells in physical memory used to hold the objects. Real computers do not operate directly on objects. Hence a correspondence between addresses generated by a process and objects must exist. And, because objects are represented in physical memory, there must also exist a

[1] The term object is used to mean a logical grouping of information, such as a page or segment. It will be defined more closely in the following section.

correspondence between object names and physical addresses. These correspondences can be found whether or not a process is executing.

Thus it can be argued that a programmer is more concerned with the manipulation of objects (as conceptual entities) than with there representation either in the process or physical address spaces. In some operating systems the naming, manipulation, sharing and protection of objects is a central concept, for example Multics [Organic, 72] and, in a most general way, Hydra [Wulf et al, 74]. A major objective of the model presented here is to be able to represent these addressing structures in a simple and natural way.

5.2.1 Objects

One of the intriguing aspects of the study of computers is that the operation of a computer may be represented in a useful way at many different levels of abstraction. For the present discussion we must carefully establish the appropriate level. We desire to represent systems at the level of the interface between an applications program and the operating system.

We have said, above, that a computation may be represented as a sequence of operations on "objects". At the operating system interface level we may define an object as follows:

> An **Object** is a logical aggregation of information which can be named to the operating system and can be directly referenced by a program. In many systems this logical grouping is called a segment.

At a higher level of abstraction, we could ignore logical groupings of information and consider a computation as operating on the entire set of information available to a process. At a lower level of abstraction we could consider individual pages, or even words, within a segment. **Pages** are physical subdivisions of segments provided to facilitate management of primary and secondary memory. They are not logical divisions which concern us at the operating system interface level. (In some systems, "pages" are not subdivisions of segments, for example, Hydra "page objects", but are independent logical entities. These can be considered fixed size segments and are Objects for our purposes.)

Associated with the definition of an object is a notion of *binding time.* A programmer (and compiler) may consider two arrays as independent objects but the linker may bind them in a single segment. During process execution they must be considered a single object because the O.S. cannot manipulate them independently.

A critical part of the definition of an Object is the phrase "can be directly referenced by a program." This phrase is intended to exclude files, in a conventional operating system, from classification as Objects. However, it is not intended to exclude entities which cannot be

accessed within a single instruction. (For example, a process under Hydra may require an operating system call to make a particular "page object" addressable because of limitations of a small address space. Similarly, to access an arbitrary word on a Nova may require first loading an index register.) We may distinguish the two cases by noting that a conventional file read operation is an operating system call which explicitly asks for an existing Object, or part of an Object, to be overwritten with new data from a file. An operation to make an Object accessible does not overwrite an existing Object, although it may make an Object temporarily inaccessible.

We will see below that the definition of an Object, when applied to real operating systems, is still not precise enough to include exactly those entities which correspond to our intuitions about segments.

5.2.2 The Three Name Spaces

Where, as discussed above, earlier writers normally have identified only two distinct address spaces; the formalism presented here has three. These are defined as follows:

1. The **Execution Environment**[1] **Name Space** is the set of names for Objects that can be generated by a process, or subpart of a process, executing on a processor. The space of names that is directly under program control[2].

2. The **System Name Space** is the set of identifiers for all unique **Objects** known to the operating system.

3. The **Physical Name Space** is the set of identifiers for each unique storage cell accessible to the system. Normally this is the set of addresses for all primary and secondary memory locations. In some systems it would include tertiary storage and cache memory.

The 'Ex-Env Name Space' is defined in the same spirit as the terms 'address space', 'virtual address space' and 'name space' are used by Denning [Denning, 70] and Habermann [Habermann, 76]. The term 'Physical Name Space' is is defined rather differently than the term 'memory space' used by Denning and others. The Physical Name Space includes all memory cells, not just those in primary (i.e. randomly accessible) memory. The 'System Name

---

[1] In most cases "Execution Environment" or "Ex-Env" can be read as as a synonym for "process". However, in some cases we will speak of multiple Execution Environments within a single process.

[2] Note that this definition includes addresses which may require multiple machine instructions to generate an arbitrary address. For example mini and micro computers without an absolute addressing mode (eg. Nova) or C.mmp where the program first presents an index in the CPS and later presents an offset within the page.

Space' is simply the identifiers[3] for all the the logically distinct objects that could be directly referenced by any process in the system.

With many operating systems the System Name Space does not exist in any explicit form. For example, consider the Burroughs B5500 operating system, the MCP. The logical entities, or 'objects' in this system are segments. Thus the 'System Name Space' is the set of names for each distinct segment. However the MCP does not support a way of directly naming an arbitrary segment. All segments could be found and named by first finding and naming each process and then finding and naming each segment within the process. However, any segment shared by more than one process would have multiple names.

A similar situation occurs in Multics. Within a process, each 'known' segment is identified by a segment number [Daley & Dennis, 68]. But, two processes may use the same segment numbers for different segments and different segments numbers for the same segment[1]. This situation is illustrated in Figure 5-1. Each process has a private segment table containing segment descriptors. Process A uses the index <i> to access a segment, labelled L. A different index value, <j>, is used by process B to access the same shared segment. However, process B uses the index value <i> to reference a different segment, M. Hydra maintains a direct listing of all objects[2] based on unique names derived from a clock.

5.2.3 Mapping Between Name Spaces

The addressing structure of any conventional system may be represented as shown in Figure 5-2. There is a mapping

$$f_1: EE \rightarrow S$$

where names generated by an Ex-Env Name (EE) are translated to the unique identifiers used by the system to name each object (S). This is followed by a second mapping

$$f_2: S \rightarrow M$$

where the system names of objects are translated to give the physical location of the

---

[3]Although an attempt will be made to carefully distinguish between the name of an object and the object itself, there may be occasions where the reader will have to rely on context - as with most programming languages.

[1]However, it appears that every segment can be uniquely identified, via a directory structure, by a 'full path name'. Also segments are tagged with a unique ID derived from the system clock. No directory is maintained on the basis of this tag - it is used to consolidate multiple references to the same segment.

[2]In this context we are only concerned with 'objects' which can be directly referenced by a user process, hence we need only consider Hydra 'Page Objects'.

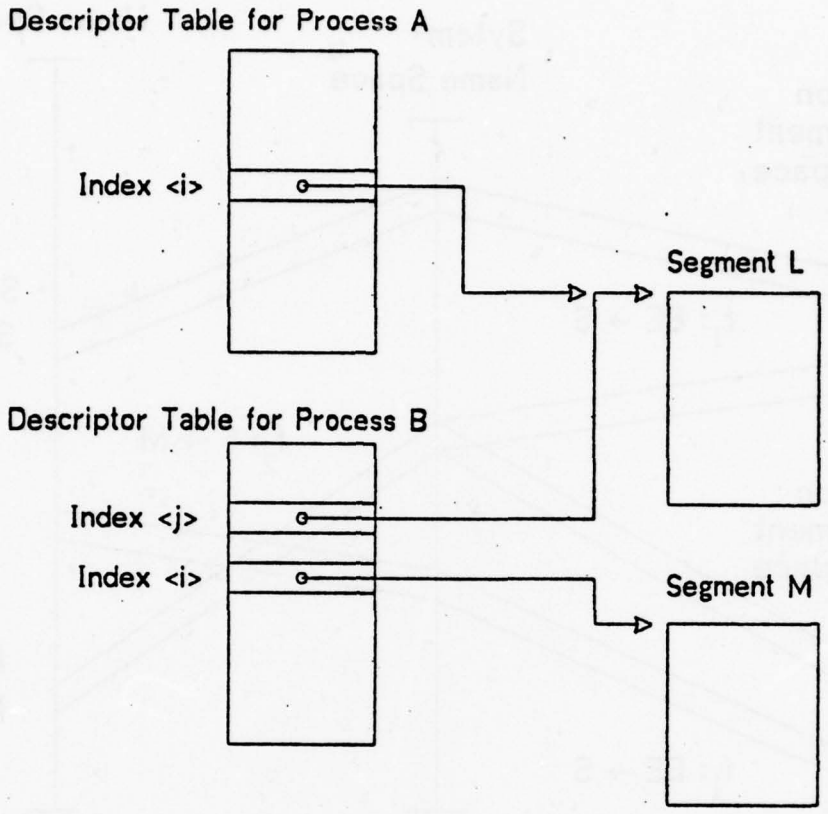


Figure 5-1: Segments without Unique Names

object.

This model is not intended to necessarily reflect the implementation of any specific addressing system. It is, however, intended to reflect the logical operation of all such schemes. One of the objectives of this chapter is to to show that a proper understanding of of the logical operation of addressing structures - particularly as viewed by programmers - will give a better appreciation of the issues in implementing specific systems. Unlike the model used by Denning, this representation of addressing mechanisms is valid for all processes, all of the time, not just for the single process which is executing. Even if a process is dormant, and partially or wholly migrated[1] to secondary storage, the

[1] Obviously, the S → M mapping may not be fully defined during the actual transfer of an object.

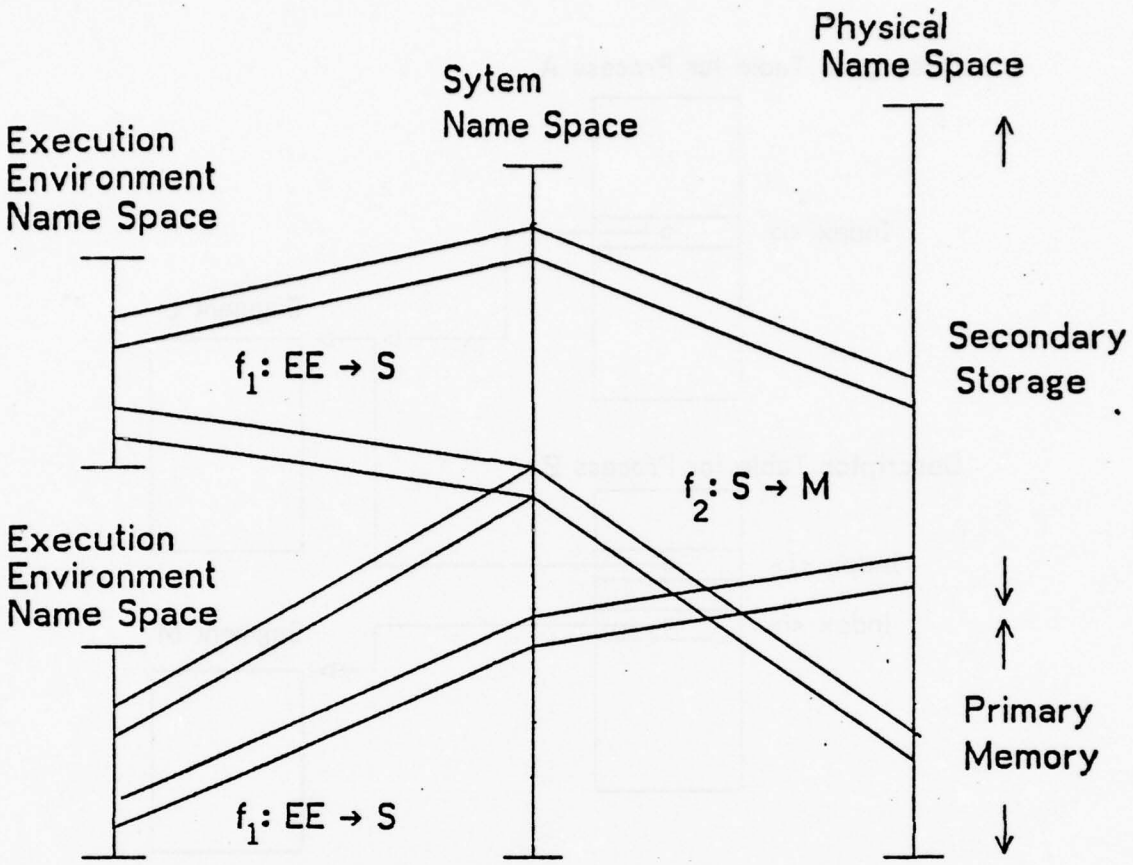


Figure 5-2: The Three Name Space Model of Addressing Structures

correspondences indicated in Figure 5-2 can be found.

5.2.4 More Definitions

We will see in subsequent sections that an important measure of the power of an addressing architecture is the number of Objects that can be addressed by a process. This is defined as follows:

> The *Ex-Env Name Space Size* is the number of objects an Execution Environment can reference.

Other characteristics of addressing architectures include:

The *Immediate Ex-Env Name Space* is the collection of objects that can be directly referenced by an Execution Environment without executing additional instructions to make an object addressable. (For example, under Hydra on C.mmp. a process has direct access to 8 objects.)

The *Physical Subdivision Size* is the size (in words or bytes), or range of sizes, for the allocation of contiguous physical address for the smallest physical subdivision of an object. This is the page size, or on an unpaged system it is the range of segment sizes.

The *Number of Physical Divisions per Object* is the range of number of pages comprising a segment.

5.2.5 Representing Mappings by Table Lookup

Without loss of generality, any function over a discrete domain can be defined by a table. Figure 5-3 shows Figure 5-2 redrawn with functions f_1 and f_2 represented by translation tables. Each table entry in the EE → S mapping corresponds to an object, hence the number of entries in each Process-to-System Name Space translation table equals the 'Ex-Env Name Space Size'. In the mapping from System Name Space to Physical Name Space (S → M) there may be multiple table entries for each segment to allow the independent placement of pages.

To relate this model more closely to real computer systems, it must be noted that special hardware is often provided to optimize address translation for an active process. Normally this hardware, invoked for each memory reference of an executing process, performs a direct translation from the Ex-Env to Physical Name Spaces. That is, there is a single level of address translation for dynamic references. There is no intermediate reference to the System Name Space. (The consequences of this lack will be discussed in Section 5.7.) This direct mapping, shown in Figure 5-4, is the functional composition, $f_1 \cdot f_2$, of the f_1:EE → S and f_2:S → M translations. In terms of the table-lookup representation, the two tables have been merged into one. This procedure is clearly valid if and only if the entries in the EE → S and S → M tables remain unchanged. Changes can usually only occur in very controlled ways: A change of the EE → S mapping would normally require an explicit request by the process to the operating system. (For example, an overlay operation.) A change in the S → M mapping corresponds to physically moving an object, normally to or from secondary storage. In either case the operating system gains control of the processor and can, where necessary, ensure a rebinding of the direct EE → M mapping on the basis of the new EE → S and S → M mappings. Note that on a multiprocessor it may be necessary to interrupt every processor to ensure that all hardware mapping registers are updated. (This issue will be pursued below.)

5.3 The Purpose of Address Mapping

The mappings performed between the three name spaces may have several independent

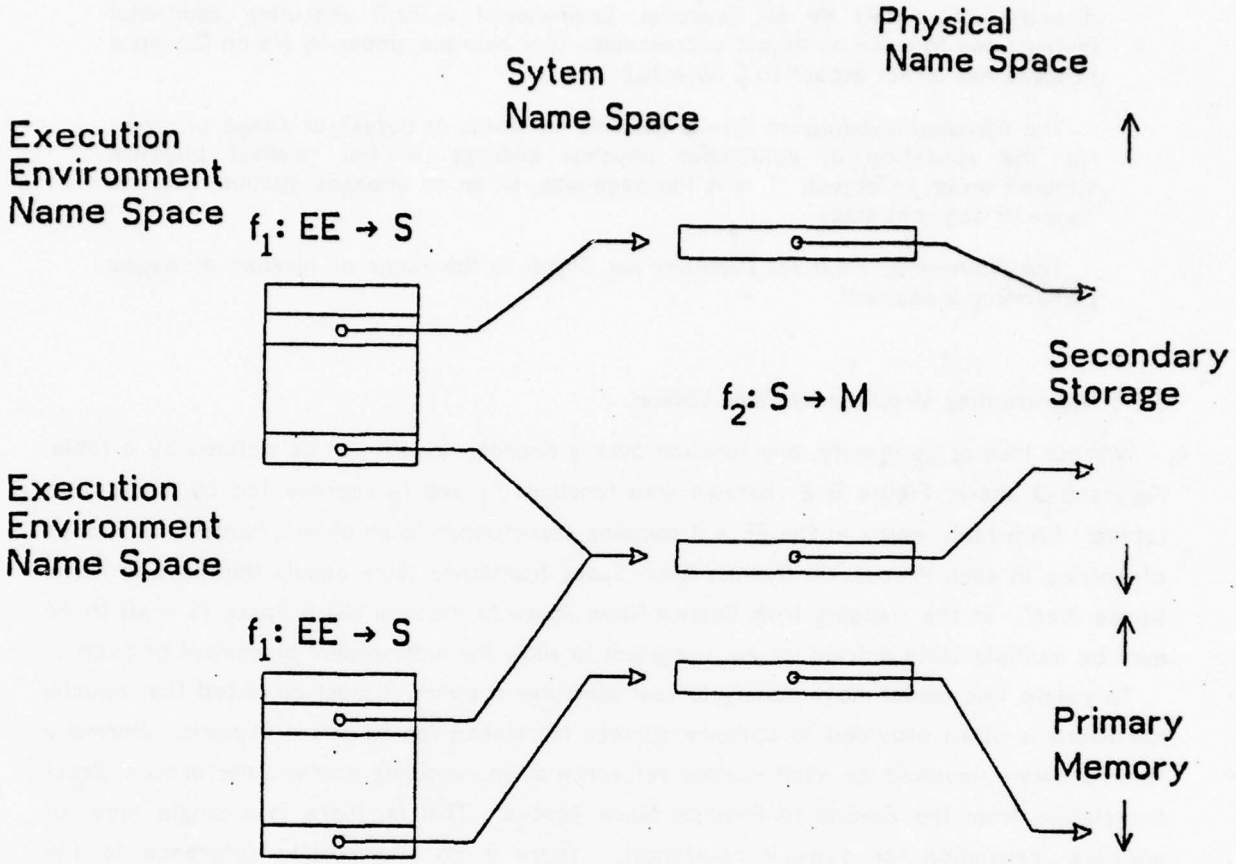


Figure 5-3: The Three Name Space Model Implemented with Translation Tables

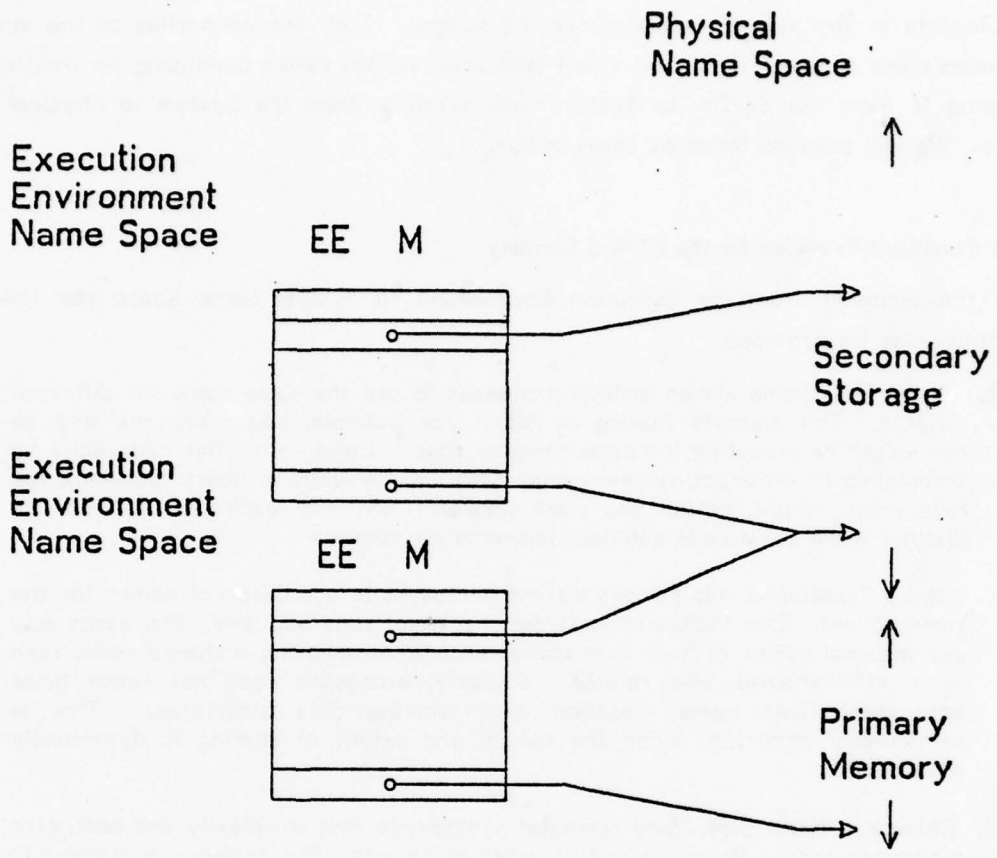


Figure 5-4: Direct Execution Environment → Physical Address Mapping

properties:

1. The most obvious property is the translation of names between name spaces. A correspondence is established between names with (potentially) different numerical values in different name spaces.

2. The mappings provide the opportunity for a change in name size between name spaces. That is, the range of valid names (e.g. address size in bits) may be different for each name space.

3. The mapping tables provide an opportunity for invoking control mechanisms. A control operation may be invoked either when a mapping is established or when a particular mapping is exercised.

The three properties outlined above are independent; they may occur separately or in combination in any specific address mapping scheme. They are properties of the mapping between name spaces. Thus their effect (and applicability) varies depending on whether the mapping is from the Ex-Env to System name space or from the System to Physical name space. We will examine these six cases in turn.

5.3.1 Functions Provided by the EE → S Mapping

In the mapping from the Execution Environment to System Name Space the following functions may be provided:

1. **Name Translation** allows distinct processes to use the same name for different objects. This permits sharing of code. For example, two processes may be concurrently executing the same compiler code. Names within this code must be translated to different system names according to which process is making the reference. Input, output and stack segments, etc. for each process must be distinct while some code and data segments are common.

   2.**Name Translation** also permits distinct processes to use different names for the same object. This facilitates both the sharing of code and data. Processes may use internal names of their own convenience when invoking a shared code, such as a standardized Sine routine. Similarly, processes need not reach prior agreement over name allocation when sharing data structures. This is particularly important when the nature and extent of sharing is dynamically determined.

2. **Change in Name Size.** Many computer systems, to limit complexity and cost, give a process access to only a small number of objects. For example, a process in the TOPS-10 system can access only two segments. However, the System Name Space must normally be larger, being equal to the sum of all the distinct objects addressable by all processes in the system.

   3.A related, but distinct, use of name expansion property is for systems where the address range of a process is less than the primary memory size[1]. This allows a large primary memory to be exploited by having many co-resident processes.

   4.A third possibility is when a process can name more objects than actually exist. This is true in Multics, for example. Reasonable responses to the naming of a non-existent object including causing an error trap or considering it as a request to create an object of that name.

3. **Control Mechanisms.** Most systems which provide protection on the access to objects implement that protection in the mapping mechanism from the Ex-Env

---

[1]This is a distinct case, because a computer system may allow a process to address all of primary memory yet only name a small number of objects. For instance, a process may have access to a single segment with a 32 bit address. Name size expansion would take place when translating to the System Name Space, but it would not be related to exploiting a large primary memory.

name space to the System Name Space. The protection is usually imposed at two levels. Whether a process is permitted any form of access to an object is usually checked at the time a mapping to that object is established. (In a capability based system, establishment of a mapping corresponds exactly to giving a process a capability for the object. In Multics, it corresponds to creating a segment descriptor for the object.) The type of access permitted is usually also specified at this time. The second level of protection is when a reference is actually made and the type of operation (e.g., read/ write) is checked as the mapping is exercised. Note that protection is imposed at this level because the EE → S mapping is specific to both the process and the object.

4.Another use of the control property when establishing this mapping is to trigger the migration of an object from secondary storage to primary memory. This form of "paging" is discussed further below, with respect to the S → M mapping.

5.3.2 Functions Provided by the S → M Mapping

In the mapping from the System Name Space to the Physical Name Space the following functions may be provided:

1. **Name Translation.** The separation of system names from physical addresses allows objects to be independently placed in primary memory. This leads to better memory utilization, and allows dynamic creation and deletion of objects without direct regard for primary memory addresses.

2. **Change in Name Size.** In one sense the number of objects represented in the System Name Space must equal the number of objects in the Physical Address Space. (There is a one to one mapping.) However, in conjunction with the **Control Mechanism** property of this mapping, a change in name size is used to give a compact address for objects resident in primary memory. That is, the System Name Space may be much larger than the number of objects potentially co-resident in primary memory. Conversely, the address range for objects in secondary storage may be larger than the System Name Space. Thus address contraction can be used for objects resident in primary memory while address expansion is used for objects in secondary storage.

3. The **Control** property of this mapping is used to permit special action to make an object accessible when it is on secondary storage or to allow movement of an object out of primary memory. Broadly, there are two "paging strategies": demand paging, where page presence is checked on each access, corresponds to a control mechanism invoked each time the address mapping is exercised. With "request" or "programmed" paging, a page is brought in at the explicit request of a program-- usually the act of making an object directly addressable. That is, this form of paging is invoked by a control mechanism associated with establishing an EE → S mapping.

4. The initial **establishment** of an S → M mapping corresponds to the creation of an object.

5.4 Expansion of the Execution Environment Name Space

We have already discussed the use of name size expansion in the EE → S mapping operation (Section 5.3). In some systems, particularly those based on mini or microcomputer architectures, it is necessary to expand the addressing range of a single process beyond that supported directly by the architecture.

Frequently this is done by subdividing the immediate address space of the processor into sections (often called pages). The program can then "overlay" this Immediate Ex-Env name space with objects drawn from the full Execution Environment name space. This method of address expansion is shown in Figure 5-5. The Ex-Env name space is expanded by using high order bits of the immediate (processor generated) address to index a table of Name Expansion Registers. The effective Execution Environment address is formed by concatenating the contents of the selected Name Expansion Register with the low order bits of the immediate address.

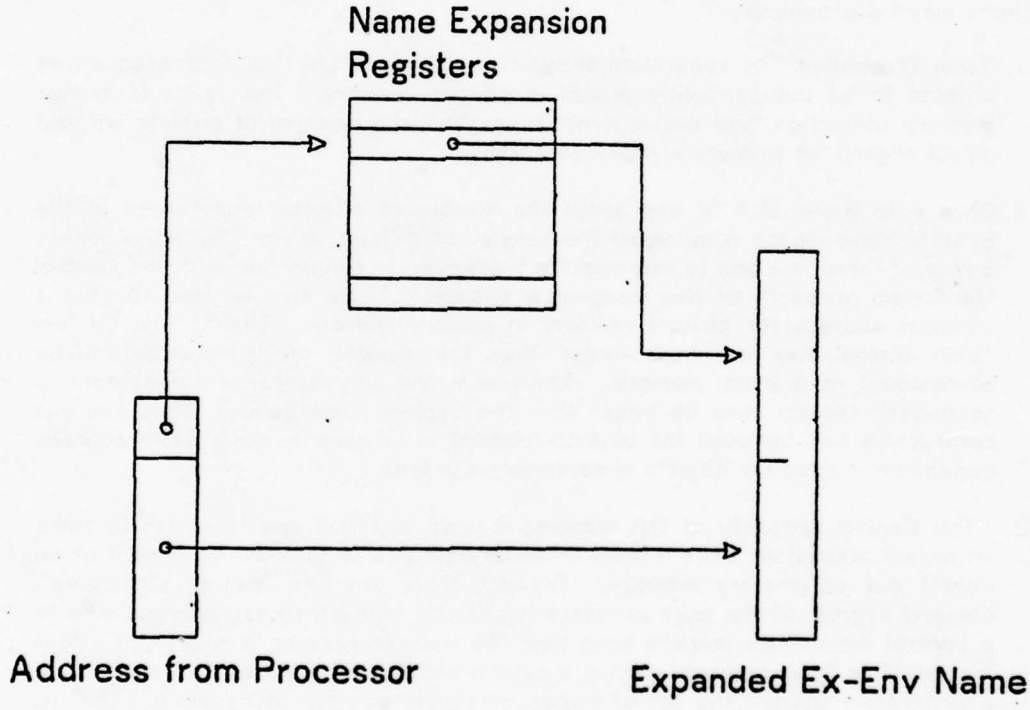


Figure 5-5: Expanding the Execution Environment Address Space

The total number of objects accessible to the process is increased because the contents of any Name Expansion Register can be changed, conceptually under direct program control, at any time. Thus the Name Expansion registers provide a level of address mapping which is completely under program control. Logically this is similar to other forms of address mapping provided within the basic processor instruction set, for example, indexing indirection, etc.

In practice, the intermediate references to the Name Expansion Table and other mapping tables do not take place for every memory reference. They are subsumed into a single level of translation performed by hardware relocation tables.

It is worthwhile to note that in most commercial machines relocation registers (and more generally, base-limit registers) are not provided for the purpose of expanding the address space of an individual process.They are provided to allow independent placement of segments or pages in primary memory and to allow exploitation of large physical memories by having multiple processes in memory concurrently. (For example, PDP-10's and PDP-11's have relocation registers and primary memory address spaces larger than the basic address size supported by the architecture--yet most operating systems for these machines do not support expansion of the address space of an individual process.)

The Ex-Env name space expansion mechanism described here is generally acknowledged to be unsatisfactory [Newell and Robertson, 75; Marathe, 78]. We will briefly examine some of the shortcomings.

5.4.1 Unwanted Bindings of the Name Expansion Registers

As already noted, the Name Expansion registers are selected by the high-order part of an immediate address. This method of selection leads to a number of restrictions on the use of address expansion mechanisms. Note that in this context, discussion of the Name Expansion registers applies directly to the relocation registers used in most implementations.

The appearance of an address in the code or data part of a program directly binds the Name Expansion register to be used to access the corresponding object. This binding can only be overcome by masking out the high-order bits of the address and or-ing in the index of a different Name Expansion register. Typically, this imposes a minimum overhead of three instruction executions.

The use of an internal pointer within a data structure binds the Name Expansion register to be used to access the data structure. Thus, a decision made at the time of creation of the data structure restricts (Name Expansion) register allocation for all programs that will access that data structure (assuming that the address masking overhead is intolerable). This kind of

interdependence is clearly undesirable programming practice. It also makes it difficult and expensive to automate management of the Name Expansion registers (i.e., management of the processor's immediate address space) within a general purpose high level language.

The problem is compounded when a program requires simultaneous access to several similar data structures. When there are large numbers (more than the likely number of free Name Expansion registers) of instances of the same type of data structure there is little option except to bind all instances to the same Name Expansion register. When accessing two such data structures, for example performing a comparison or a matrix multiplication, a program must either repeatedly reload the single designated Name Expansion register or do address masking for access to one of the data structures.

The programming difficulties associated with this form of address expansion are not an inevitable consequence of using a processor with a short word size. Many of the problems are avoided in architectures which use a field within an instruction to select a Name Expansion register. (See Figure 5-6.) This approach, which must be part of the original processor architecture and not an "add on", leaves register binding to the compiler or assembly language programmer. Register usage is not bound within data structures. Standard techniques for general purpose and index register allocation can be used.

5.4.2 Performance Overheads of Name Expansion

Programs which execute on processors with a small immediate address space, but require access to a large address space, may incur substantial performance overheads due to managing of Name Expansion registers. Marathe [Marathe, 78] shows that approximately 18% of the memory references within the kernel of Hydra are concerned with saving and restoring of relocation registers. In part this is due to small number of Name Expansion registers which are available to be changed. Of the eight relocation registers, five are permanently allocated (local memory, I/O page, stack, fixed code, fixed data) and of the three remaining, one is used to overlay code and two to overlay data.

Within the kernel of Hydra, and in other operating systems, the addresses used are physical addresses and there is no question of protection. Thus the operating system can manipulate the contents of relocation registers directly to gain access to the full physical address space. User programs cannot be permitted to directly manipulate relocation registers because: (1) names from within an Execution Environment must be translated to physical addresses, and (2) other processes must be protected from erroneous or malicious behavior. For a user process, the act of changing Name Expansion registers must invoke a protected name translation and a change in the hardware mapping registers. In the initial implementation of Hydra, with PDP-11/20 processors, a change of addressability required a

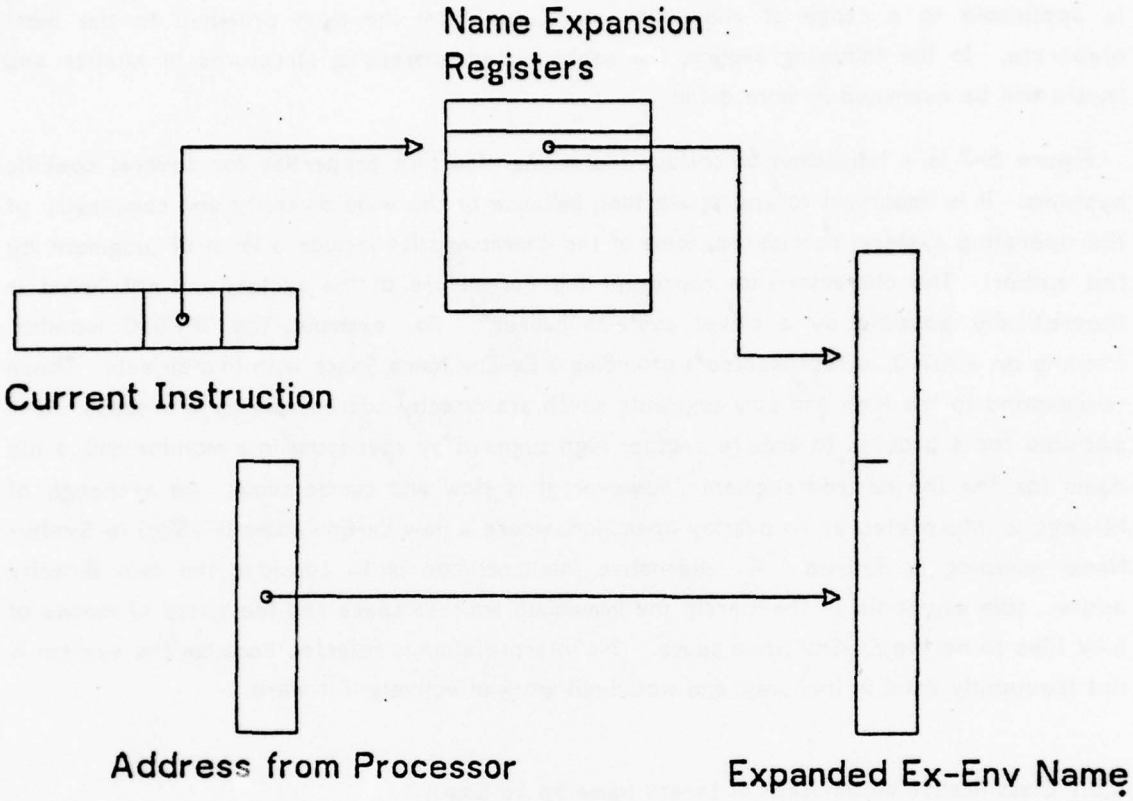


Figure 5-6: Name Expansion Register Selected by Instruction

trap into the kernel and incurred a 1 ms overhead. A comparable operation is required in FAMOS [Habermann et al, 76]. The present implementation in Hydra, using PDP-11/40's, is microcoded and requires approximately 20 us.

Although difficult to measure, it is probably fair to say that the prime disadvantage of this form of name expansion is the burden placed on programmers and the extreme difficulty of automating the use of name expansion registers within a high level language. The mechanism is very prone to programming errors and makes programs difficult to debug.

5.5 Examples of Address Mapping in Real Systems

In this section, the abstract model developed above will be applied to several common

systems. These examples will round out the description of the model and demonstrate that it is applicable to a range of addressing structures from the most primitive to the most elaborate. In the following section, the sopnisticated addressing structures of Multics and Hydra will be examined in more detail.

Figure 5-7 is a tabulation of certain addressing structure properties for several specific systems. It is important to appreciate that, because of the wide diversity and complexity of the operating systems considered, some of the characteristics include a level of judgment by the author. The characteristics represent the normal use of the system and not "what is theoretically possible by a clever systems hacker". For example, the TOPS10 monitor, running on a KA10, is represented a providing a Ex-Env Name Space with two objects. These correspond to the High and Low segments which are directly addressable by a process. It is possible for a process to acquire another High segment by specifying in a monitor call, a file name for the the desired segment[1]. However, it is slow and cumbersome. An exchange of Hi-segs is interpreted as an overlay operation, where a new Ex-Env Name (Hi-Seg) to System Name mapping is defined. An alternative interpretation is to consider the two directly addressable segments as the merely the immediate address space and the space of names of SAV files to be the Ex-Env Name space. This interpretation is rejected because the system is not frequently used in that way, and would not work effectively if it were.

5.5.1 Classification on the Basis of Ex-env Name Space Size

Despite the apparent diversity of the machine-operating system combinations considered, these addressing structures can be broadly ranked along a single dimension. A grouping according to the size of the Ex-Env Name Space gives a good correspondence with a programmer's intuitive evaluation of the power of the addressing structure. The systems have been grouped according to the number of objects in their Ex-Env Name Space. A second independent dimension is the physical subdivision of objects. Paged systems, and those with small segments, are obviously more convenient for management of primary memory. (Memory management, because it does not directly effect the programmability of systems and is well presented in the literature, is not discussed here.)

Groupings, according to Ex-Env Name Space Size, are:

Primitive systems have no address mapping hardware. Each process has access to a single Object, usually called a "Core Image". These systems require objects to be loaded in fixed addresses in primary memory. Code sharing can only be achieved by defining programming

---

[1] One application is to minimize primary memory usage. This is done by the Bliss/11 compiler. Used in this way it reflects a user implemented paging (i.e. physical subdivision) of a single object, the combined set of code.

MAPPING HARDWARE MACHINE OPERATING SYSTEM	Bare PDP-11	Base -limit	Dual base-limit KA-10 Tops-10	Dual base-limit paged KL-10 Tops-10	8 Relocs PDP-11 RSX-11M	B5500	Multics	PDP-11 FAMOS	C.mmp Hydra	Cm* Staros
* of Objects in the Execution Environment Name Space.	1	1	2	2	8	Large	Large	Large	Large	Large
* of Objects directly addressable by processor. (Immediate Ex-Env Sixe)	1	1	2	2	8	Large	Large	8	8	15
Objects may be Contiguous in processor address space.	—	—	YES	YES	YES	No	No	YES	YES	YES
Object Sharing between different processes.	No	No	Only Hi Seg Shared.	Only Hi Seg Shared.	YES	No ?	YES	YES	YES	YES
* of Objects in Name Space fixed at Ex-Env Creation.	YES	YES	YES	YES	YES	YES	No	No	No	No
Process Independant Naming of Objects	No	No	No	No	No	No	only via Directory	YES	YES	YES
Automatic Directory lookup of string names of Objects.	No	No	No	No	No	No	YES	No	No	No
Dynamic Creation and Passing of Objects	No	No	No	No	No	No	No ?	YES	YES	YES
No. Physical Divisions / Object	1	1	1	1 -256	1	1	Large	1	1	1
Size Physical Divisions (words)	32 K	—	128K	512	32 - 4K	1 - 1K	64, 512	32 - 4K	4K	1 - 2K
Independant Placement Physical Divs. in Memory	No	YES	YES	YES	YES	YES	YES	YES	YES	YES

Figure 5-7: Some Addressing Structure Characteristics

conventions for the use of the Process and Physical Name Spaces.

Simple systems are implemented with a base displacement register (and often a limit register for protection). These systems allow a process access to a single Object which may be placed anywhere in memory. The primary application of this kind of addressing structure is to allow simple timesharing and multiprogramming. An individual process appears to have a private machine while the operating system has the freedom to allocate primary memory in a way which allows multiple processes to be co-resident. Sharing is not possible.

Standard systems (such as Tops10, RSX11-M) which allow a small fixed number of segments which are always directly addressable. These systems provide the basic characteristics desired, as noted by Denning and others, for any reasonable addressing structure:

1. Objects may be placed anywhere in primary memory (in units of the physical subdivisions) independent of the address used by the process.

2. Objects may shared, eliminating the need to have duplicate copies of editors etc. in memory.

3. Different processes may use different names for the same shared object and likewise the same name may refer to different objects for different processes. (For example, this allows multiple compiler incarnations with shared code. The same object name within the shared code maps to different data segments according to which process is executing.)

The support for more than one object permits code sharing but is not necessarily adequate for data sharing. Objects cannot be easily created and passed between processes. Objects, in the sense used here, may not be recognized to have an existence outside of their ownership by a process. The number of objects owned by a process is fixed at the time a process is created and any sharing requires explicit pre-arrangement. (For example, knowledge of the global name of the Fortran compiler.)

Sophisticated systems, for example Multics and Hydra, allow a process access to a large[1] number of objects. These systems have the desirable properties noted above for *Standard* systems, but in addition the sharing of small units of code and data is greatly facilitated. Objects can be dynamically created and passed between processes. Objects are recognized to have an existence beyond the lifetime of the process that created them. This makes it possible to provide pools of objects for message systems, IO mechanisms, etc.

---

[1] In this context, large means large enough that no reasonable program is in danger of exceeding the limit.

5.6 Creation of Addressing Environments

In addition to the characteristics discussed above, *Sophisticated* Addressing systems also support a sequence of addressing environments generated during the execution of a process. The separate execution environments (protection domains, subprograms, etc.) usually represent independently compiled (and often independently written) program modules. These program modules are provided with parameters via a call mechanism and return computed results in some manner. These addressing environments are the logical extension of the user/kernel addressing environments provided on Standard systems. The decomposition of a program into small independent modules is motivated in part by considerations of programming methodology. It also is part of a trend away from the view that a timesharing system must simply provide a large number of independent virtual machines to the view that it must support a diverse community of users, often working in close cooperation.

5.6.1 Representation of a Sequence of Addressing Environments

The addressing model presented here allows representation of a snapshot of the addressing environment of a number of processes. However, it is not fully adequate for the representation of a sequence of addressing environments generated by a single process. Despite these shortcomings it is useful for representing some simple cases. Note that when the locus of control passes from one environment to another the address mapping function $f_1: EE \rightarrow S$ is replaced with another mapping function $f'_1: EE \rightarrow S$. The mapping $f_2: S \rightarrow M$ is not effected.

5.6.2 Fabry's Classifications of Addressing Structures in Re-entrant Programs

Fabry [Fabry, 74], in his important paper on capability based addressing, examines both the functional and implementation characteristics of a variety of addressing structures with respect to changes in environments and sharing between processes. Because this paper represents one of the few attempts to critically examine addressing environments it is worthwhile to review his classifications. An alternative view will be offered in a subsequent section.

For convenience, Fabry's Figures 1 through 6 have been reproduced as Figures 5-8(1:4) and 5-9(5:6).

Fabry considers the example of two processes concurrently executing a program called 'MAIN'. (See Figure 5-8(2).) The program includes a call to a subroutine called 'SUB'. Each process requires a private segment for local data, called 'DATA'. Fabry examines addressing

Fig. 1. Segment addresses.

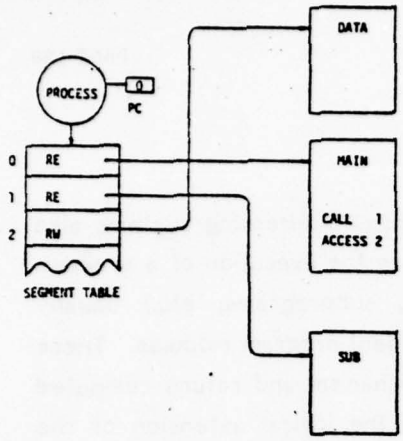


Fig. 2. Shared segment addresses.

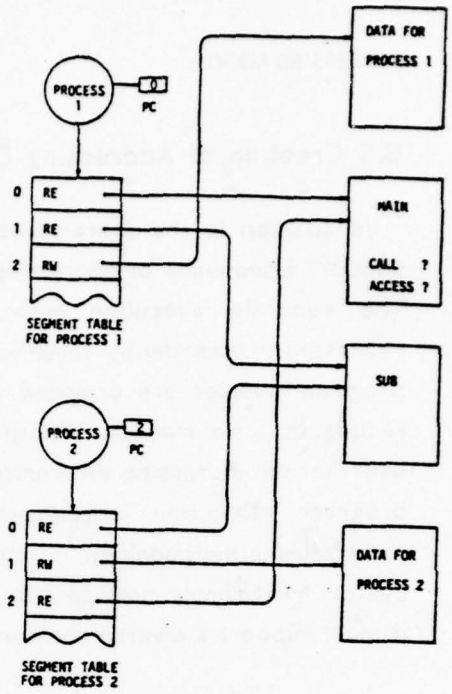


Fig. 3. Uniform address solution.

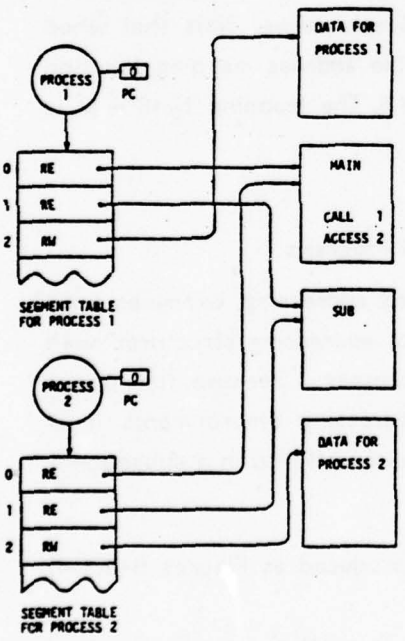


Fig. 4. Indirect evaluation solution.

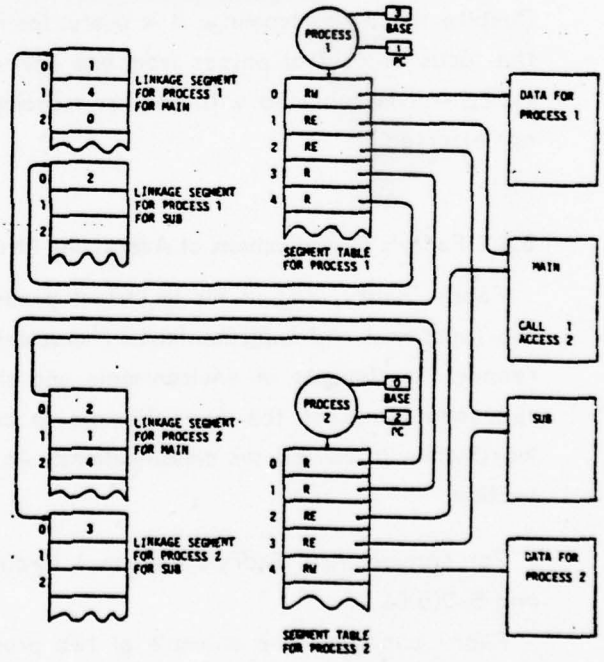


Figure 5-8: Fabry's Figures 1:4

Fig. 5. Multiple segment table solution.

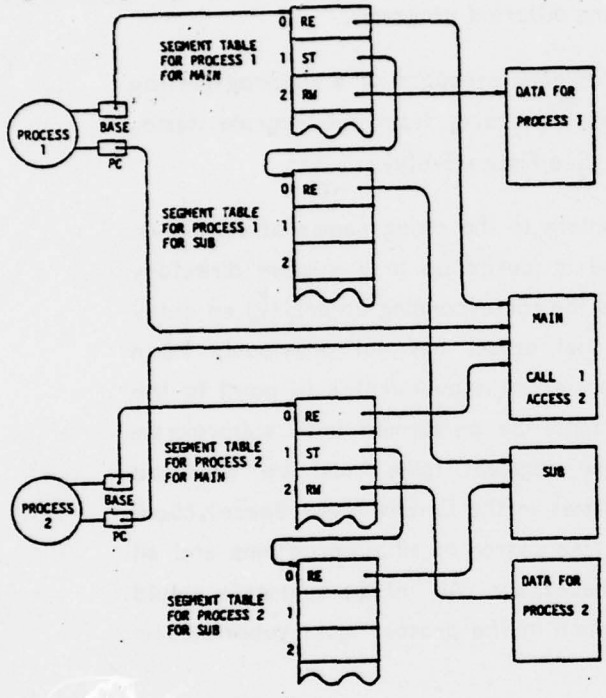


Fig. 6. Capability addressing solution.

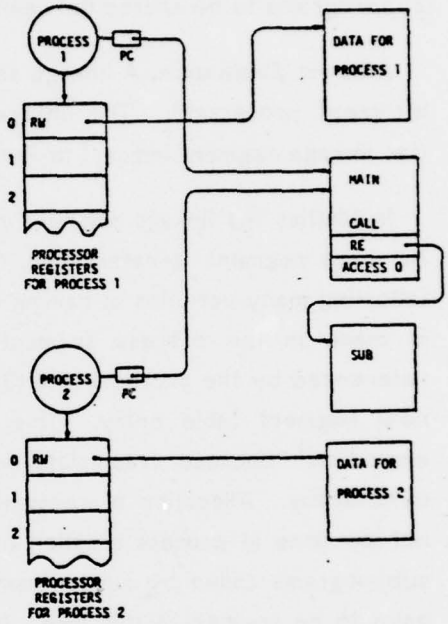


Figure 5-9: Fabry's Figures 5:6

structures which will allow a single copy of the program to be shared by multiple processes, i.e. how to allow program re-entrancy.

Uniform Addressing Solution. Here a single addressing environment exists for the duration of the process execution. (See Figure 5-8(3).) Naming conflicts in subprograms are avoided by simply forcing the user to compile, or at least link, the entire program at once. This permits multiple incarnations of the same program to share code but does not permit subprograms to be shared between processes executing different programs.

Indirect Evaluation. A linkage segment is created for each execution of a subprogram (by different processes). The linkage segment provides a mapping from subprogram names (i.e. linkage segment indices) to Process Space Names. See Figure 5-8(4).

In Multics the linkage segment initially contains pointers to the string names of segments. As each segment is referenced, (1) the string name is looked up in a system directory (allowing many varieties of naming conventions) to find the corresponding object, (2) an entry is made in the process segment table (assuming that object has not previously been referenced by the process), and (3) the linkage segment entry is overwritten to point to the new segment table entry. This kind of binding cannot be performed until subprogram execution[1] because free slots in the process wide segment table must be allocated dynamically. Allocation of segment table slots (i.e. names in the Ex-Env Name Space) could not be done at process creation time because a full tree search of all subprograms and all subprograms called by subprograms etc. would be necessary. All linkage segments would have to be created at this time. In the normal execution of the process most subprograms may never get called[2].

In Multics, subprograms can reference any segment in the current process segment table and are not restricted to indirect references via the linkage segment. Thus there is no inherent protection when crossing a subprogram boundary even though the addressing environment has been modified. To provide protection, the Multics ring structure is

---

[1] In Multics binding is performed only the first time a segment is referenced within a subprogram. All bindings could be performed at subprogram entry but this may mean some references are resolved unnecessarily.

[2] Except for whatever savings result from only resolving references that are actually executed, this scheme involves the same number of name bindings that would conventionally be done with a linker (or on the B 5500, with the compiler) prior to execution. We will see below that if subprograms are executed within an addressing environment independent of the calling process then many linkage operations can be avoided.

introduced[3] (Roughly, segments created in a lower value rings are inaccessible except for function invocations.)

Multiple Segment Table Solution. Although similar, this differs from the Indirect Evaluation solution in that the linkage segment becomes a private segment table for each invocation of each subprogram. (See Figure 5-9(5).) Fabry points out that there is now no process wide address space, and asserts that this makes parameter passing difficult. Fabry does not point out that a major advantage of this structure is that it provides full protection across subprogram boundaries (in contrast to the multics structure).

Capability Addressing. This is Fabry's preferred mechanism, see Figure 5-9(6). In this scheme, absolute pointers to segments (i.e. capabilities) may be embedded directly in a program and in the registers of a processor. The notation "ACCESS 0" means access via the capability in register 0. Fabry's representation of this solution, relative to the other solutions he describes, is misleading. He asserts "this scheme does away with segment tables and with mandatory indirect evaluation of shared addresses"[1]. The mechanism, as shown, is far less general than the other solutions he describes.

1. No addressing environment change is indicated when invoking "SUB". How will references to local data within "SUB" be handled ? This cannot be embedded in the code because it differs for each invocation. Presumably he assumes "SUB" will use space inherited from "MAIN" for local variables - implying the kind of cooperation between program modules that many systems attempt to avoid.

2. Access to "DATA" for each process is shown as being via a program accessible machine register. Fabry allows capabilities to be held in machine registers in a protected way. He claims that the use of registers is qualitatively different from use of a segment table:

> "The allocation of processor registers is under control of the person or compiler generating even the smallest section of code; one is always free to redefine the use of these registers by saving the contents and later restoring them. Thus there is no requirement for for a central mechanism to define the use of the registers, and the

---

[3]Multics may be taken as an example of the central importance of the addressing architecture in the design of a computer system. We have seen that design decisions in the Multics architecture have led to a requirement for two elaborate operating system mechanisms not required by later systems with similar goals. The need for execution-time binding of all program references in Multics is a direct consequence of the decision to allocate all descriptors in a single process-wide segment table. We will see below that allocation of descriptors on a sub-program basis allows far greater flexibility in binding time and can avoid the overhead of many redundant dynamic bindings. The ring structure protection scheme is, at least in part, a consequence of permitting a sub-program to access any segment accessible by the process. Both of these design decisions in turn relate to the method for passing parameters to a sub-program invocation.

[1][Fabry,74] page 407.

main problem with the uniform address solution is avoided[2]."

This appears to beg the question. The compiler (or programmer) must explicitely load the register with a capability for a process specific "DATA" segment. This code is shared and cannot be specific to a particular process. Therefore the capability must be loaded from a data structure private to each process but with a standard format for interpretation by the shared code. This amounts to a segment table, which he claims to have avoided.

5.6.3 Process Wide or Subprogram Limited Addressing ?

Although all Sophisticated Addressing Structures allow a sequence of addressing environments to be created by a process, they may be classified according to whether an addressing environment is viewed as associated with a process or with an invocation of a subprogram. Multics and CAP[Needham and Walker, 77], for example, provide a per process segment table. Subprograms, invoked in the course of process execution, have access to some subset of the objects in the segment table. In CAP, a capability list for each subprogram invocation, is provided to translate indices in the segment table; this provides protection within the process. The linkage segment mechanism in Multics provides the same name translation function, but subprograms may make direct references via the process wide segment table. Protection is provided by modifying the segment table entries when crossing protection ring boundaries [Organic, 72]. These schemes are represented approximately by Figure 5-10.

An alternative to maintaining a per process segment table is to maintain a segment table for each invocation of a subprogram. Since this table is private to each invocation of a subprogram, it can translate directly from names used with the subprogram to object names in the System Name Space. (The entries in this table are often called Capabilities.) The additional level of indirection through a process wide segment table is no longer necessary. This approach is represented by Figure 5-11. Systems which use this approach include Hydra, FAMOS [Habermann et al, 76], Staros [Jones et al, 77], CAP-3 [Herbert, 78].

The alternative systems represent different operating system design philosophies. In the former systems, the fundamental executable unit is a process. Subprograms invoked by the process are seen as large, Algol-like subroutines which (in Multics at least) may inherit addressability of objects from the caller. The latter systems, with subprogram based addressing, are derived in part from programming methodology which advocates the modular decomposition of programs. Communication between modules is minimized and made very explicit to emphasize their interdependencies.

The fundamental executable unit manipulated by the operating system is now an activated

---

[2] [Fabry, 74] page 407.

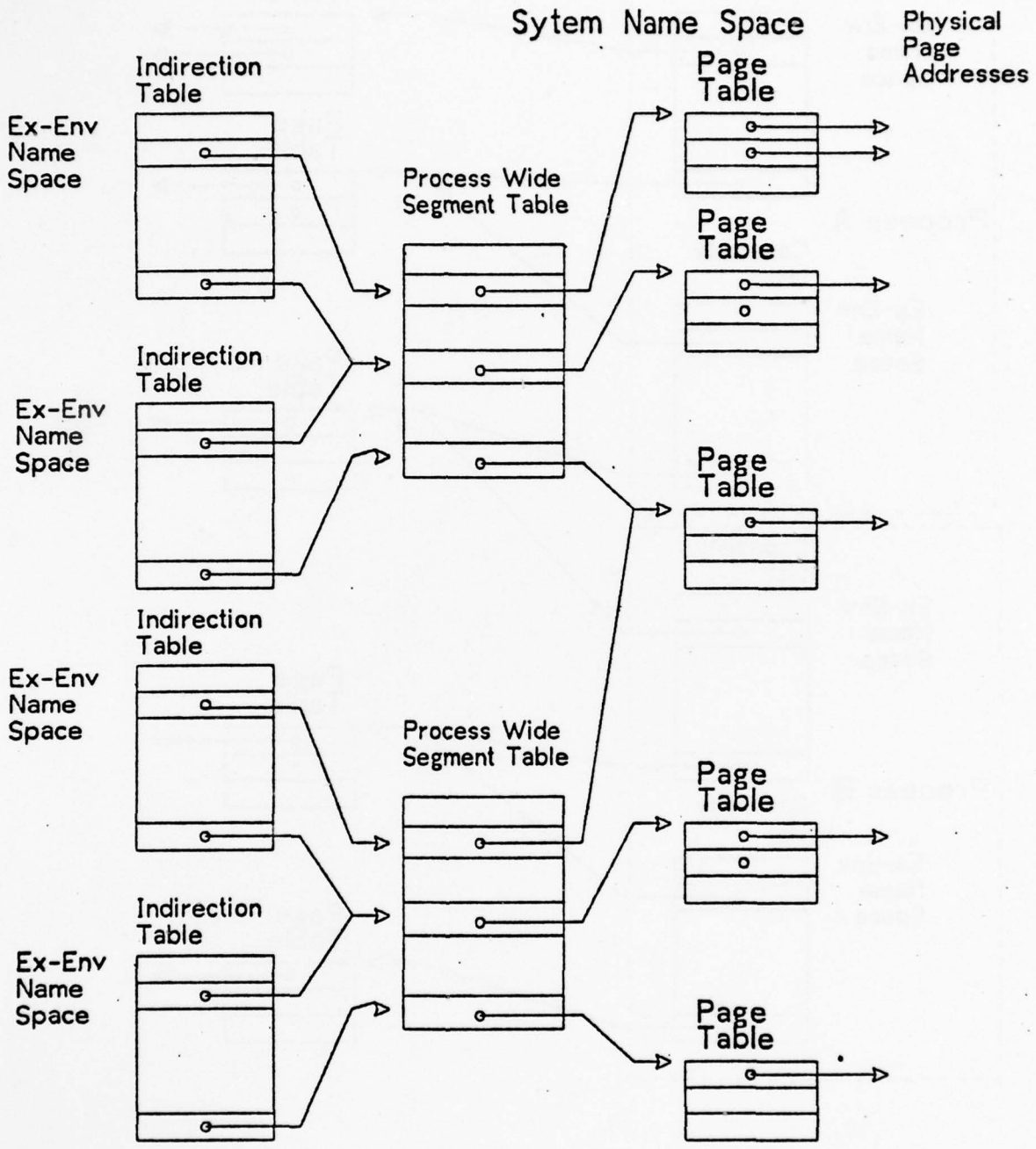


Figure 5-10: Addressing using a Per Process Segment Table

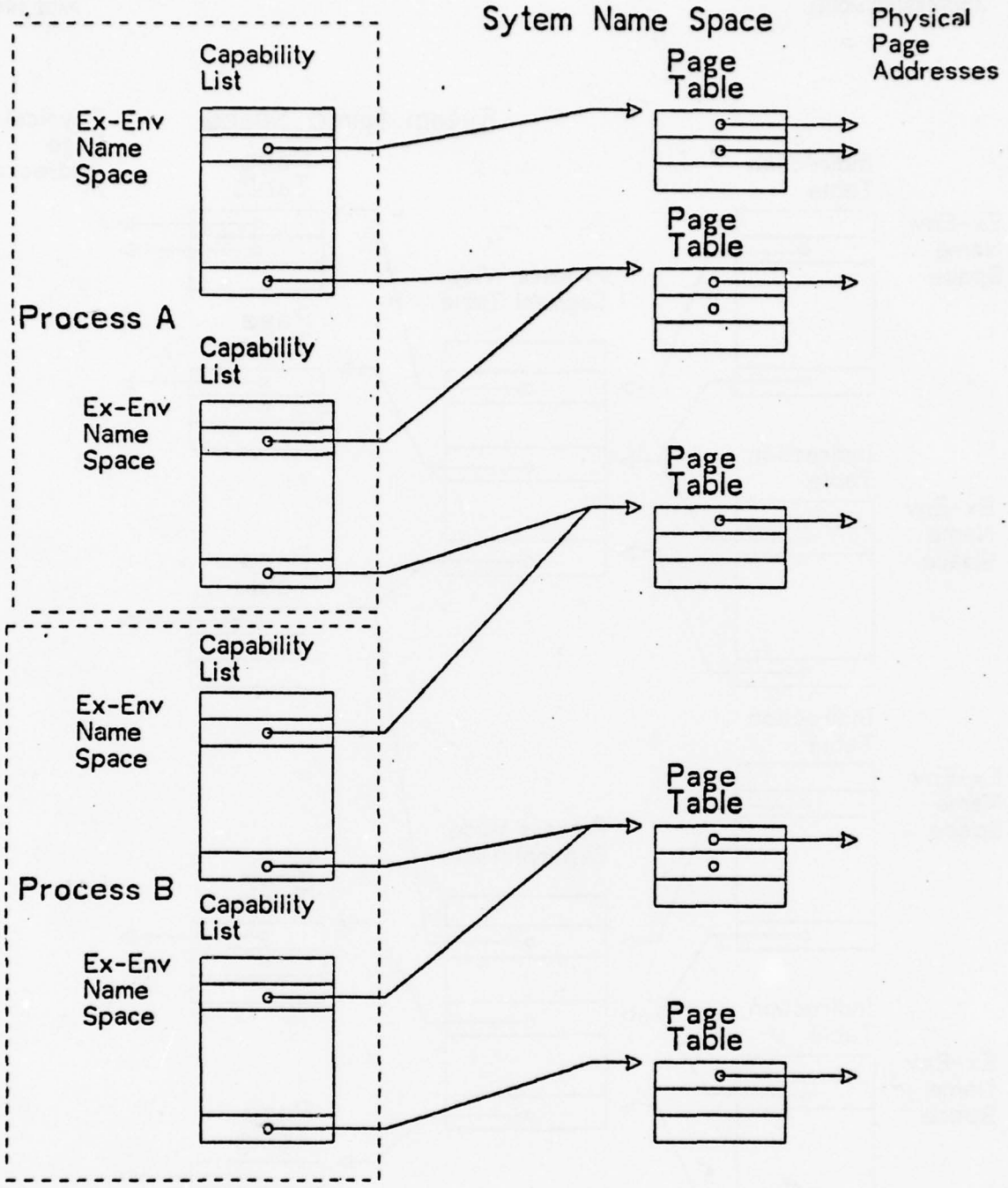


Figure 5-11: Use of Capability Lists in a Segmented, Paged System

program module. Parameters passed during invocation are bound into the activated module and there can be no other dependency between the called and calling modules. (Independent of parameters, modules may statically share access to common data structures and other subprograms.) A process can now be represented as a chain of activated modules. Forking may result in a process being represented as a tree structure. In systems with a process wide segment table, provision for multiple subprograms to be spawned in parallel within a process may be very awkward. Rodger Needham[1] cites, as an advantage of the CAP two level addressing structure (a capability list per subprogram activation, where capabilities contain indices for a process wide segment table), that capabilities may be copied and passed as parameters within a process without maintaining reference counts to the segments[2]. This is also true in Multics, where in this context a capability is just an integer representing a process wide address. The reason that reference counts are unnecessary in CAP is that it is assumed that all segments private to a process will be deleted when the process terminates. The problem is not avoided for segments which are shared between processes.

There are several reasonably clear advantages for having addressing environments associated with program modules rather than processes:

1. Addressing information (i.e. a segment table or capability list) which is common to all activations of a module need be defined only once. This includes all code, own data, and references to other program modules. In Multics, a directory must be searched and a slot in a process wide segment table allocated for every segment referenced during process execution. With subprogram based addressing environments, only those segments which are private to a particular invocation (local data etc.) need be dynamically resolved. There is also a corresponding saving of space because mapping information is not duplicated for each incarnation of a module. This may be an important consideration with some frequently used system functions[3].

2. The mechanism for passing parameters between subprograms of the same process and between processes can be uniform.

3. The mutual independence and small size of subprogram addressing environments makes them suitable for expressing and exploiting parallelism in a multiprocessor.

4. In otherwise comparable systems, the process wide segment table adds an additional level of indirection for each memory reference. (In most systems, however, this would be bypassed by some form of caching in hardware.)

---

[1] Talk given at CMU, October, 1977.

[2] Reference counts on segment descriptors are usually used to facilitate deleting segments when they are no longer accessible.

[3] Needham gives the example of of the memory allocator which must contain segment descriptors for all segments in main memory. Similar problems arise with any module which has access to a large data base.

Note that 1, above, implies that the addressing environment of a module is broken into at least two components. A static part, which may be shared by all activations of a module, and a dynamic part which is private to each activation and contains capabilities for local data, parameters, etc.

5.6.4 Some Conclusions about the Creation of Environments

As we have seen the creation of addressing environments depends on the properties of compilers, linkers and operating systems. The two most important activities are the allocation of names within a subprogram and the binding of these names to objects in the System Name Space. In all cases the allocation of names is done by the compiler (and possibly completed by a linker) because the programmer's intentions are clear at this level. The various mechanisms differ in the time at which subprogram names are bound to objects. For example:

1. In Fabry's *Uniform Addressing Solution* the names of objects containing code are all bound at compile time (and may be shared), the names of private data objects are not bound until program activation.

2. In Multics all names are bound at the time the name is first used during execution.

3. One of the important advantages of subprogram based addressing environments is that it allows flexibility in binding time. References to pre-existent objects which will be common to all instantiations (i.e. references to other subprograms, non-local data, etc.) can be bound at subprogram creation - the earliest possible time. This binding need not be repeated for each activation of the of the subprogram. References to private data and parameters can be bound at subprogram activation. Name binding can also be delayed until the last possible moment, immediately prior to the first reference, or, in the extreme, rebound for each reference.

There are numerous strong motivations for decomposing systems into a large number of independent modules: programming methodology, protection, ease of software maintenance, potential for parallel execution on a multiprocessor, etc. However, in practice the cost of establishing a new addressing environment when invoking a program module limits the grain size of program decomposition. A better understanding of addressing mechanisms should lead to more efficient implementations of protected subprogram invocation with existing machines, and provide a solid foundation for the hardware implementation of high level primitives in new machines.

5.7 Implementation of Sharing

We have used the a three name space model to represent the addressing structures of a wide range of systems. However, we have also noted that the addressing hardware in most conventional machines actually implements a direct mapping from the Ex-Env name space to the physical name space. There is no intermediate reference to the system name space. In Simple systems, where objects cannot be shared, the Ex-Env and System Name Spaces may be merged[1] without loss of generality. This is not possible when an object is shared between processes because either: 1) there would be multiple System Names for the same Object (a contradiction) or 2) the flexibility of processes using different names to reference the same object would be lost (thus sharing would require prior arrangement on name allocation) and the use of the same name to reference different objects (as with shared compilers and editors) would be impossible. **The responsibility therefore falls on the operating system software to provide the logical effect of a three name space mechanism with hardware which directly supports only two.**

5.7.1 Sharing of Pages in Hydra

Hydra [Wulf et al, 74] has a conceptually pure and very general notion of an object and mechanisms for naming objects. The unit of memory allocation, a Page-Object, is merely an instance of a large and extensible class of typed entities that are protected and maintained by the operating system. A Hydra-object is referenced by a Capability which contains the global name of the object and specifies which operations may be performed on the object by the possessor of the capability. For each distinct object the system maintains a small data structure defining the type of the object and pointers to its components. In the case of Page-objects, the only Hydra-objects we are directly concerned with, the object contains a "presence bit" and the address of the page (either in main memory or secondary storage). At this level of description, the Hydra addressing structure exactly fits our three name space model. Capabilities provide the EE → S translation and Hydra-objects provide the S → M translation. Figure 5-12 illustrates this and shows the additional details of part of the processor generated address selecting an entry in the "Relocation Page Set" which in turn selects a capability from the "Core Page Set" (CPS). The remainder of the processor generated address selects a word within the designated page. The address mapping

[1] Merging is equivalent to making the EE → S mapping a direct 1 to 1 correspondence.

hardware on C.mmp is very primitive[2]. It provides user, Kernel and IO address spaces each with 8 relocation registers. The normal 64 K byte address space is divided into 8 equal size pages of 8 K bytes. Each page may be independently relocated in the 2^{25} byte primary memory address space. The three high order bits of a processor generated address are used to select a relocation register whose contents are concatenated with the lower 13 bits from the processor to form a physical address.

We must now reexamine the addressing structure of Hydra in the light of the hardware support provided. The referencing of memory is the most frequent operation a process performs and so any substantial delay, such as indirecting through memory, is considered intolerable. The differences between the conceptual addressing structure and actual implementation may be viewed as optimizations of frequently used access paths. (This is another example of the design principle of exploiting skewed distributions discussed in Section 1.4.1.) The optimizations are in the form of caching (or copying) the effective result of a table-lookup operation, or sequence of table-lookup operations. Figure 5-13, while still simplified, shows more detail of the actual addressing structure. Comparing this with Figure 5-12, the conceptual addressing structure, we note:

1. While in an inactive form a capability contains a full "Global Name" of an object. However, when used in the "Core Page Set" (and elsewhere) the "Global Name" is replaced with a direct pointer (physical address) to the "page Object". This direct pointer bypasses the directory structure for translating a "Global Name" into the address of an object. It clearly speeds up access to an object via a capability.

2. Where conceptually the "Core Page Set" contains only capabilities for "Page Objects", in practice it also contains the physical address of the page referenced by each capability. This bypasses a table-lookup operation via the "Page Object". The intent of this cached page pointer is to speed up the process of making a page addressable[1].

3. In the conceptual representation, Figure 5-12, the 3 high order bits of each address generated by the processor are used to index the "Relocation Page Set". This in turn indexes the "Core page Set", etc. (The "Relocation Page Set", because it is effectively under direct program control, is the mechanism used to give a program access to a larger address space.) Rather than evaluate this chain of indirection for each memory reference, another level of caching is provided by the relocation registers. Entries in the Relocation Registers, corresponding to the "Relocation Page Set" data structure, contain direct pointers to the pages

---

[2] It was designed in the early 1970s as a CMU add-on to the PDP-11/20 minicomputers. Given the choice of a 16 bit computer with a fixed instruction set and a desire to maintain program compatibility there is no reasonable alternative. Commercial versions, released later, provide variable size segments rather than 4 K word pages. Although this would be convenient on C.mmp it would not impact the problems discussed here.

[1] It may be that a slight redesign of the "Page Object" data structure would make the page base address accessible quickly via the capability. This would eliminate the need for the copy of the page address within the "Core Page Set".

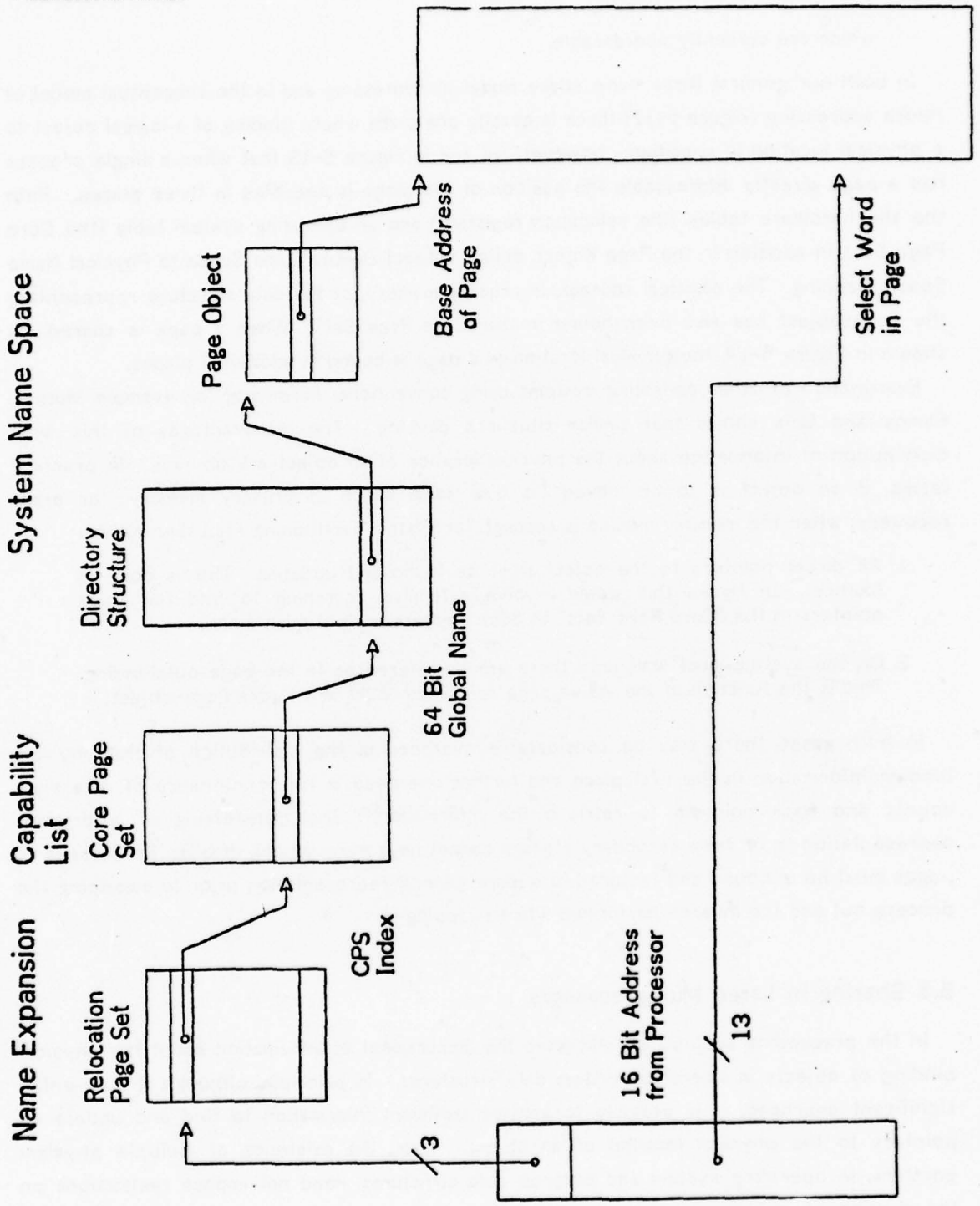


Figure 5-12: Conceptual Addressing of a Page in Hydra

which are currently addressable.

In both our general three name space model of addressing and in the conceptual model of Hydra addressing (Figure 5-12) there is exactly one place where binding of a logical object to a physical location is specified. However, we see in Figure 5-13 that when a single process has a page directly addressable the position of that page is specified in three places. Both the the hardware tables (the relocation registers) and an operating system table (the Core Page Set), in addition to the Page Object, define a direct Ex-Env Name Space to Physical Name Space mapping. The physical address, in primary memory, of the data structure representing the Page-object has also been bound in the "Core Page Set". When a page is shared, as shown in Figure 5-14, the physical location of a page is bound in additional places.

Examination of other operating systems using conventional hardware, for example Multics, Famos and Unix shows that similar situations develop. The disadvantages of this wide distribution of information about the physical location of an object are obvious. In practical terms, if an object is to be moved (to free some space in primary memory, for error recovery, when the memory module is suspect, for system partitioning etc.) then either:

1. All direct pointers to the object must be found and updated. This is done in Multics. In Hydra this would involve extensive searching to find the page pointers in the "Core Page Sets" as back pointers are not maintained.

2. Or, the system must wait until there are no references to the page outstanding. This is the function of the active page reference count in a Hydra Page-object.

In both cases there may be considerable overhead in the distribution of the physical binding information in the first place and further overhead in the maintenance of reference counts and back pointers to retrieve the information. The transferring of a process representation to or from secondary storage cannot be a mere moving of bits. References to pages must be unbound and restored to a more general representation prior to swapping the process out and the inverse performed when swapping in.

5.8 Sharing in Large Multiprocessors

In the preceeding section, we discussed the disbursment of information about the physical binding of objects in operating system data structures. In principle, although it may entail significant overhead, it is possible to provide sufficient information to find and update all pointers to the physical location of an object. Thus, the existence of multiple physical pointers, in operating system and program data structures, need not impose restrictions on the moving of objects or other operations which require mutual exclusion. This leaves operating system designers free to make tradeoffs according to estimates of the relative

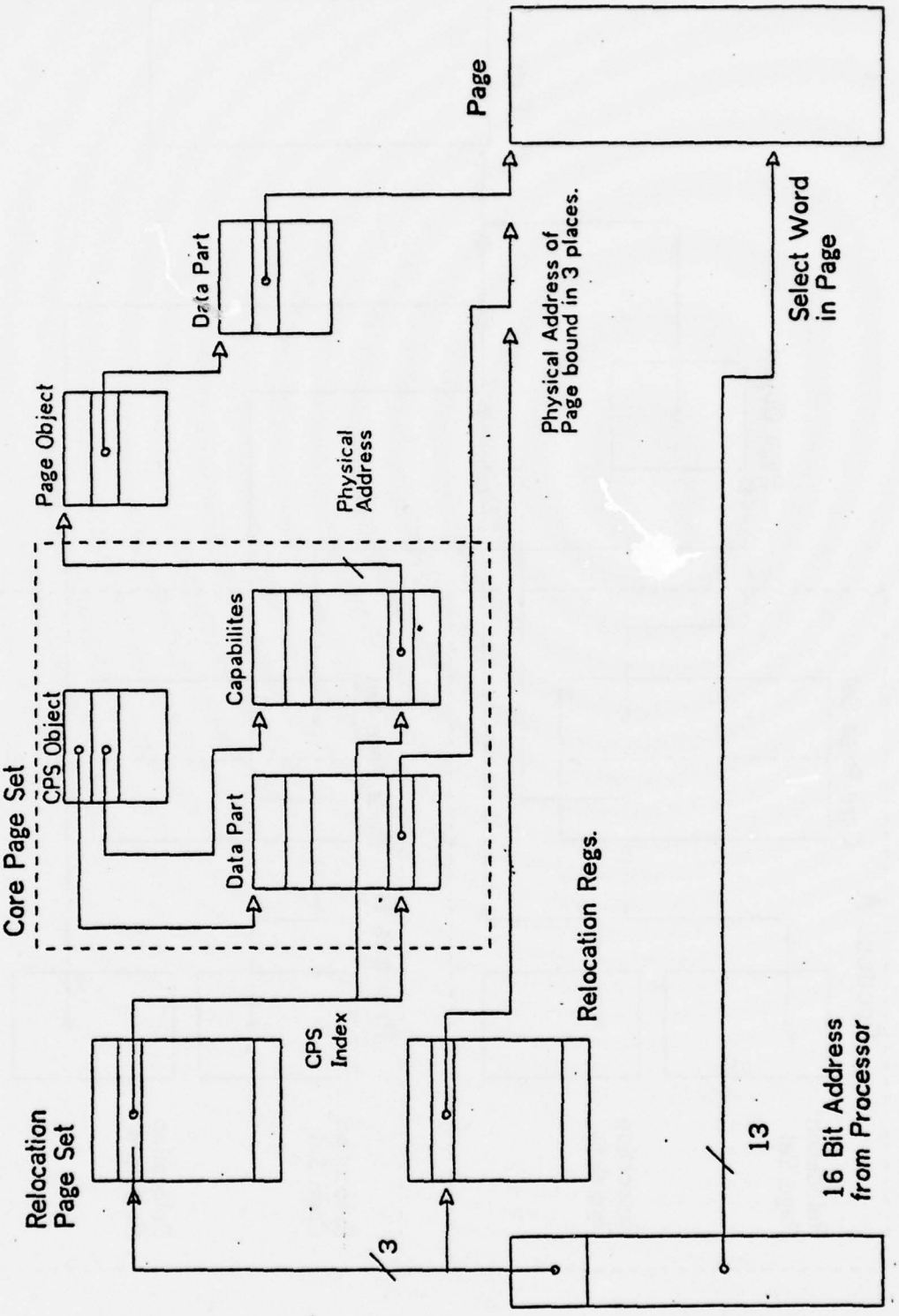


Figure 5-13: A Page Addressable by a Single Process in Hydra

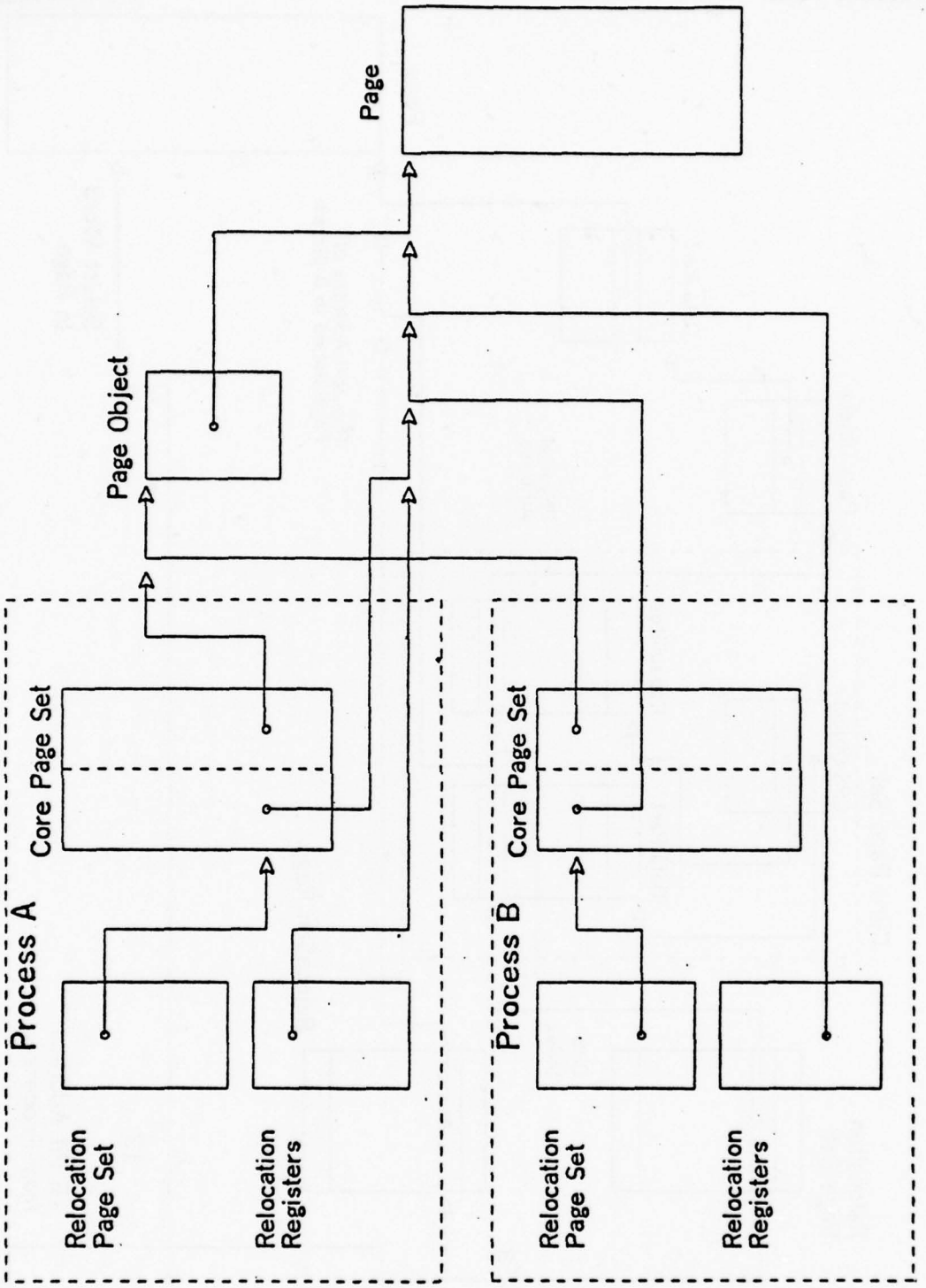


Figure 5-14: Page Shared by Two Processes in Hydra (simplified)

frequency and costs of operations which manipulate pointers to objects.

Physical pointers to objects also appear in the hardware registers of processors. These registers are not normally accessible by other processors (in a multiprocessor) and so they present a different problem from the finding and updating of pointers in main memory data structures. In this section we will discuss this problem in relation to conventional multiprocessor structures. Rather than an extensive survey or analysis, this section is intended to show that an important problem exists and to provide motivation for the following section which describes an alternate way of structuring a multiprocessor to eliminate the problem.

In Section 5.2.5, the representation of address mapping by table lookup was discussed. Figure 5-4 indicates the direct EE → M translation usually provided by the hardware in both multi and uni processors. These registers correspond to the relocation registers in C.mmp and to the associative memory used in Multics. Whenever a page or segment is directly addressable by more than one processor, its physical location will appear in multiple hardware registers. If the object is to be moved, the multiple pointers must be updated to reflect the non-accessibility of the object (to cause a page-fault trap) and, if the object is merely moved in primary memory, then the pointers should be updated to reflect the new location.

There are several obvious approaches to this updating problem:

1. Maintain reference counts and don't move objects that are currently addressable. This increases the overheads of making an object addressable. Also it does not work if the object must be moved because of suspect memory, system partitioning, or to gain performance etc. It also fails if a processor dies with an object addressable.

2. Interrupt all processors and cause all or part of their addressing registers to be updated. This requires a mechanism for establishing that all processors have updated their mapping tables. Establishing that all processors have completed an operation is probably impossible in a situation where the number of active processors changes dynamically. I.e. dead processors will never complete and it is necessary to wait for any newly active processor to complete. Also the total performance loss probably grows as the square of the number of processors[1].

---

[1] We assume that the number of object moves is proportional to the number of useful instructions executed by each processor. For example, assume that a processor requires an object moved for every C (Computed) useful instructions executed and that the cost of updating the address mapping is W (Wasted) instructions for each processor. Then for a multiprocessor, with P processors, the proportion of wasted instructions per processor is $W \cdot P / (C + W \cdot P)$. I.e. on average, each processor updates its mapping tables P times during the period it executes C useful instructions. The total processing power lost is $W \cdot P^2 / (C + W \cdot P)$. Let C = 10,000 and W = 10, i.e. 0.1% of the time would be lost in the degenerate 1 processor case. With 100 processors, approximately 9% of processing power would be wasted. With 1000, processors, approximately 50% of execution time would be devoted to updating mapping tables.

3. For each object, maintain a list of processors that can reference it directly. Then interrupt only these processors when moving the object. This increases the the overhead making an object addressable, or not addressable, but eliminates the quadratic growth of overhead noted in 2, above. Implementation of this scheme would be very messy and would have some of the reliability problems noted above.

5.8.1 Other Motivations for a Single Point of Object Definition

So far, we have discussed only one motivation for have only a single place where the physical location of an object is defined. When an object is moved, or about to be moved, all processes referencing the object will see the change on the next reference to the object. There are not multiple places to update. It may be argued that this is a comparatively rare event and that some additional overhead is acceptable. From the point of view of reliability, system reconfiguration, and general criteria for a clean design it can certainly be argued that it should be possible to move any object at any time - even if substantial cost is involved. There are other strong motivations for desiring an addressing structure where the location and other properties of an object are defined in (logically) a single place and that all references to the object refer to this object definition:

1. The object definition may contain a lock bit which is inspected on every access. This allows mutual exclusion for (normally) short multi memory reference operations. On Cm* these operations include stack operations, message and data queues, moves of two word capabilities indivisibly, primitive lock and increment-decrement operations etc.

2. The single point of object definition can be recorded in a highly redundant, reliable, protected manner. It can be used to enforce protection by only allowing specific operations. For example, only mailbox operations are permitted on Cm* mailbox segments. This can be enforced even if there is a breakdown in the capability-based protection scheme. (E.g. if a remote Kmap becomes erratic.)

3. References may be temporarily suspended and restarted later. During this extended period the object may be moved, checksumed, failsafed, have internal errors corrected, etc.

4. A single point of definition simplifies maintenance of use and dirty bits and gathering of reference rate statistics.

These examples will be explained more fully in the following chapter.

5.9 Alternative Multiprocessor Structures

The problem of updating multiple physical pointers in the hardware registers of

processors, discussed in the previous section, arose because the addressing hardware does not support the abstract model of addressing developed earlier in this chapter. Figure 5-3 shows the translation between the three address spaces implemented by table lookup. Figure 5-4 shows how the two levels of address mapping are normally implemented as a single level, direct EE → M mapping. Figure 5-15(a) indicates the position of the address mapping tables in a conventional multiprocessor. Figure 5-15(b) shows one way that the abstract addressing model could be correctly implemented on a multiprocessor. References generated by processors are first translated to System Names, with a private table for each processor, and then pass through a common table which translates System Names to the physical name space.

An obvious disadvantage of this structure, Figure 5-15(b), is the performance bottleneck caused by the single set of logic mapping from the System Name Space to the Physical Name Space. This tends to destroy the parallelism inherent in a Multiprocessor. One way to regain this parallelism is shown in Figure 5-15(c). Following a reference from a processor:

1. The name is translated from the Ex-Env to System Name Space by the table private to the processor.

2. Switch (S_A) arbitrates and separates references according to the section of System Name Space referenced.

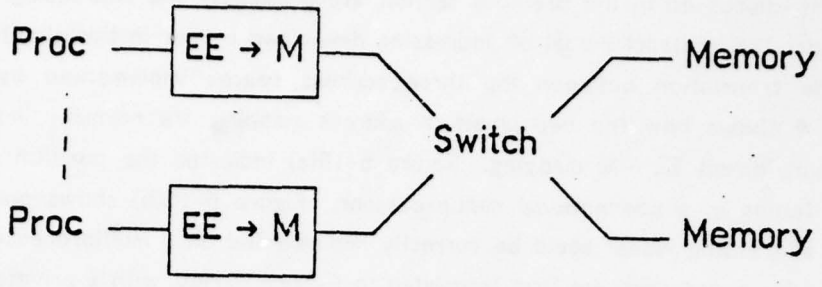
3. Separate mapping tables, for each section of the System Name Space, translate from the system to Physical Name Space (S → M).

4. The second switch (S_B) arbitrates and separates references according to the section of the Physical Name Space referenced.

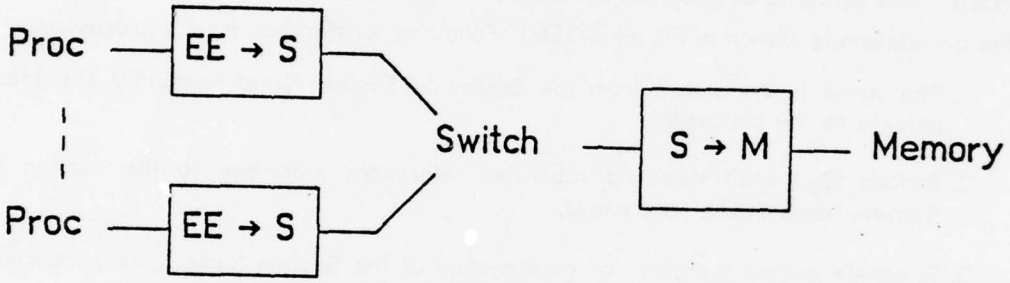
5. The reference is performed by the appropriate memory.

This structure, while offering better performance, has limitations. It requires two high performance switching structures, which is both expensive and slows access to memory. There seems to be no way of ensuring locality of switch path usage (as a basis for reducing switch cost, as in Cm*) without imposing unreasonable restraints on name space usage. Hence both switches would have to be full-crosspoints.

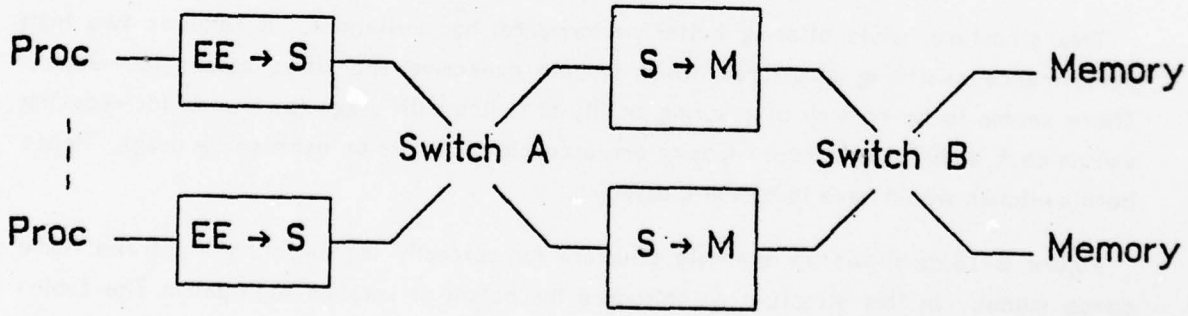
Figure 5-16(d) shows an alternate structure for correctly implementing the three name space model. In this structure we introduce the notion of **Location Anticipation**. The tables mapping from the Ex-Env Name Space to the System Name Space are augmented with additional information: the physical **Location** of the object named is **Anticipated**. This can be viewed as a partial merging of the EE → S and S → M mapping tables. A vital distinction between Location Anticipation and an actual collapsing of the mapping tables is that the location information need not be correct. The location indicated for the object may be out of



(a) Conventional Multiprocessor



(b) Common System to Physical Name Space Mapping



(c) Subdivision of S → M Mapping to Allow Parallelism

Figure 5-15: Alternative Multiprocessor Structures (a:c)

gate. Consider the operation of the structure shown in Figure 5-16(d):

1. A reference from the processor is translated by the processor's private mapping table to the System Name Space and an Anticipated Location is given.

2. The indicated location is used for an address through the switch.

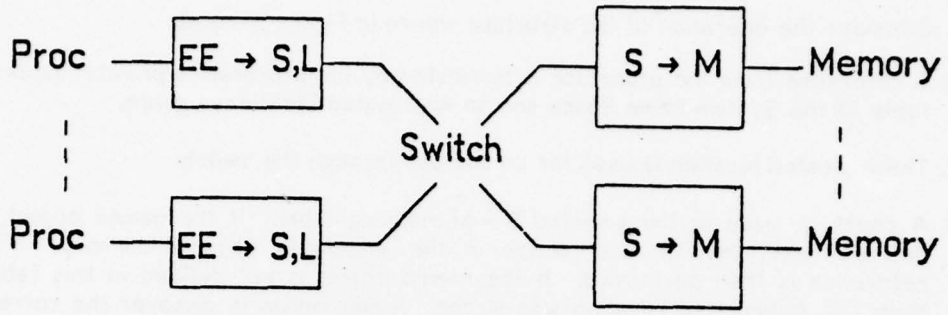
3. A check is made in the selected S → M mapping table. If the named object is defined here, then it must reside in the associated physical memory. The reference is then performed. If the named object is not defined in this table, then the Anticipated Location was wrong. A mechanism to discover the correct location must now be invoked. Once located, the memory reference can take place and the Anticipated Location information can be updated.

In the case when the Anticipated Location is correct, the memory reference proceeds quickly, with only a single level of switching. There are numerous possible ways of handling the case when the Anticipated Location is incorrect. This should occur infrequently; it is highly analogous to a page-fault in a conventional system. The most straightforward way to resolve the incorrect location fault is to have the requesting processor, or some other processor on its behalf, interrogate operating system maintained tables to find the actual location[1].

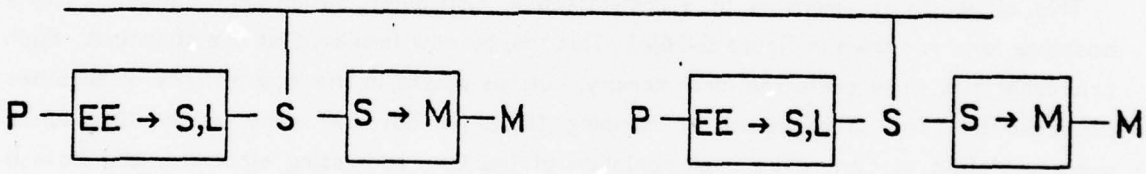
This structure is amenable to a cost-effective, distributed switch implementation. One possible form is shown in Figure 5-16(e). This the, by now familiar, Cm* like structure. Each processor has some preferred local memory, but has access to the local memory of all other processors. The arrangement of mapping tables is not the same as in the actual implementation of Cm*. The implementation of the Cm* addressing structure, and how it relates to this "ideal" structure will be examined in the following chapter.

---

[1]The operating system maintained location tables need not exactly reflect the true locations--which are defined by the contents of the S → M tables. That is, the tables in main memory need not change indivisibly with the hardware registers, but obviously they must eventually reflect the correct locations.



(d) Distributed S → M Mapping using Location Anticipation



(e) One Form of Distributed Switch

Figure 5-16: Alternative Multiprocessor Structures (d:e)

6. The Addressing Architecture of Cm*

The effective use of a multiprocessor, particularly with a large number of processors, requires close cooperation between the processors. This cooperation requires support for communication and sharing at every level of the structure, architecture and operating system. In this chapter we will see how the addressing architecture of Cm* supports cooperation and sharing between processes.

6.1 The Target Operating System Structure

No effective system design can take place without a reasonable understanding of how the system is to be used. Nor can a design be appreciated without a knowledge of its intended use. In Chapter 2, we discussed the goals of Cm* as a basis for the presentation and understanding of the hardware structure. Correspondingly, in this section, we will briefly discuss the nature of the software which this architecture was designed to support.

6.1.1 Modular Decomposition

The notion of modular decomposition is an important concept to emerge from software methodology. Parnas [72] gives criteria for determining module boundaries. In essence, he points out the advantages of hiding details of a data structure, or algorithm implementation, from other software components using the data structure, or algorithm. This abstraction of function from implementation is the basis of the present interest in Abstract Data Types [Flon, 77; Liskov and Zilles, 74]. The desirability of decomposing large software systems into small modules, with very explicit and well defined transfer of information across module boundaries, appears to be well accepted.

Software methodology aside, the advent of multiprocessors gives a strong motivation for decomposing systems into modules. On a multiprocessor, the prime objective is to find decompositions which will allow concurrent execution of separate modules. Many software engineering criteria, for example minimizing module interdependencies and explicit parameter passing, carry over directly to multiprocessors.

The hardware structure of Cm* lends itself - or perhaps more accurately - requires the decomposition of large programs into modules. In Chapter 4, we saw that (in the absence of caches for code) it is very inefficient for a processor to execute non-local code. Primary memory is a major component of system cost and so only a limited amount[1] can be provided for each processor. Thus it is necessary that large sections of code be broken into modules

[1] 28K words in the 10 processor version, the 50 processor system will have 64K words per processor.

allocated to distinct processors. An important criterion for the decomposition is to minimize calls between modules, because of the inevitable software overheads. However, data can be shared between modules with comparatively little cost.

6.1.2 Cm*--A System for Small Cooperating Software Modules

The primary model for a computation executing on Cm* is a collection of cooperating software modules. The code for a software module is normally, but not necessarily, contained within a single computer module. There may be more than one software module assigned to a physical module. Because the memory expense of a software module is primarily for its code, it is reasonably to have multiple incarnations of a software module which share code and multiplex a processor. To allow effective parallel execution of a module, it is necessary to duplicate the code in the local memory of each processor.

6.1.3 The Influence of FAMOS

The Family of Operating Systems project, FAMOS [Habermann et al, 76] provides an existence proof and model, for a system based on cooperating software modules. At the software module level, the architecture of Cm* was strongly influenced by FAMOS. FAMOS makes a natural and clear distinction between the "static" aspect of a module, which is common to all incarnations, and the "dynamic" part which is private to each incarnation (the stack, local data, etc.). FAMOS also provides a simple but powerful parameter passing mechanism to other modules, a list of capabilities for segments. These concepts have their direct counterpart in the Cm* architecture, although the implementations are very different.

6.2 Hiding the Non-Uniform Hardware Structure

Relative to conventional computers, even large multiprocessors like C.mmp, Cm* is a complex hardware structure. At each level, a major objective of the design has been to hide this complexity. At the bottom level, Cm* is a network of microcomputers. The network aspect has been hidden; the message protocol is entirely in hardware and microcode. This transforms the structure into a multiprocessor. A processor uses the same protocol to access its own memory as any other memory in the network.

At the architectural level, it is highly desirable to hide the non-uniform nature of the physical address space. As much as possible, programmers, compilers and linkers, etc. should not have to consider the physical structure. The programmer should see a uniform address space which is independent of a particular processor or cluster. The actual binding of logical addresses to physical memory will certainly still affect performance. However, the binding

can be specified independent of program generation, and usually can be done automatically by the operating system.

6.3 Major Primitives in the Addressing Structure

The unit of addressability, and the primary entity supported by the architecture, is a **segment**. It corresponds directly to the notion of an object defined in Chapter 5. Segments may be of variable size, from 1 to 2K words. Segments are not subdivided into pages. Segments in Cm* differ from conventional segments, such as those in Multics or the B5500, in that they are typed objects. Up to eight different operations may be defined for each segment type. Examples of special segments are control stacks, data queues and mailboxes. This will be discussed further in Section 6.8.

6.3.1 Capabilities

There is always an inherent conflict between providing protection and allowing sharing. Shared access to data is essential for many parallel algorithms; thus it is necessary to support memory sharing between software modules. To provide the finest grain of control over sharing and to allow efficient run time protection checking, all references to segments are via **capabilities**. A capability both names a segment and specifies which operations are permitted on the segment. It also specifies which operations are permitted on the capability; for example, can it be copied?

Capabilities are usually kept in segments of type capability list. We will refer to these as C-lists, for short. While segments within Cm* are typed or tagged, individual words are not tagged. Thus to maintain the integrity of the protection scheme it is necessary to ensure that a user program does not have direct access to a capability. A second reason for disallowing the mixing of capabilities with other data is that garbage collection of segments would be much slower. The garbage collector would have to scan all data for pointers to segments, not just C-lists, as in the present case.

6.3.2 The Structure of a Software Module

The notion of a software module, as the entity dispatched on a processor, is supported directly by the architecture. It is called an **Environment** and is shown in Figure 6-1. There are many potential hazards in defining such a high level structure as the representation of a process in the architecture. Any fixed representation may severely constrain the operating system design. Coping with an unsuitable representation may entail more complexity and overhead than was saved by supporting the structure in the architecture.

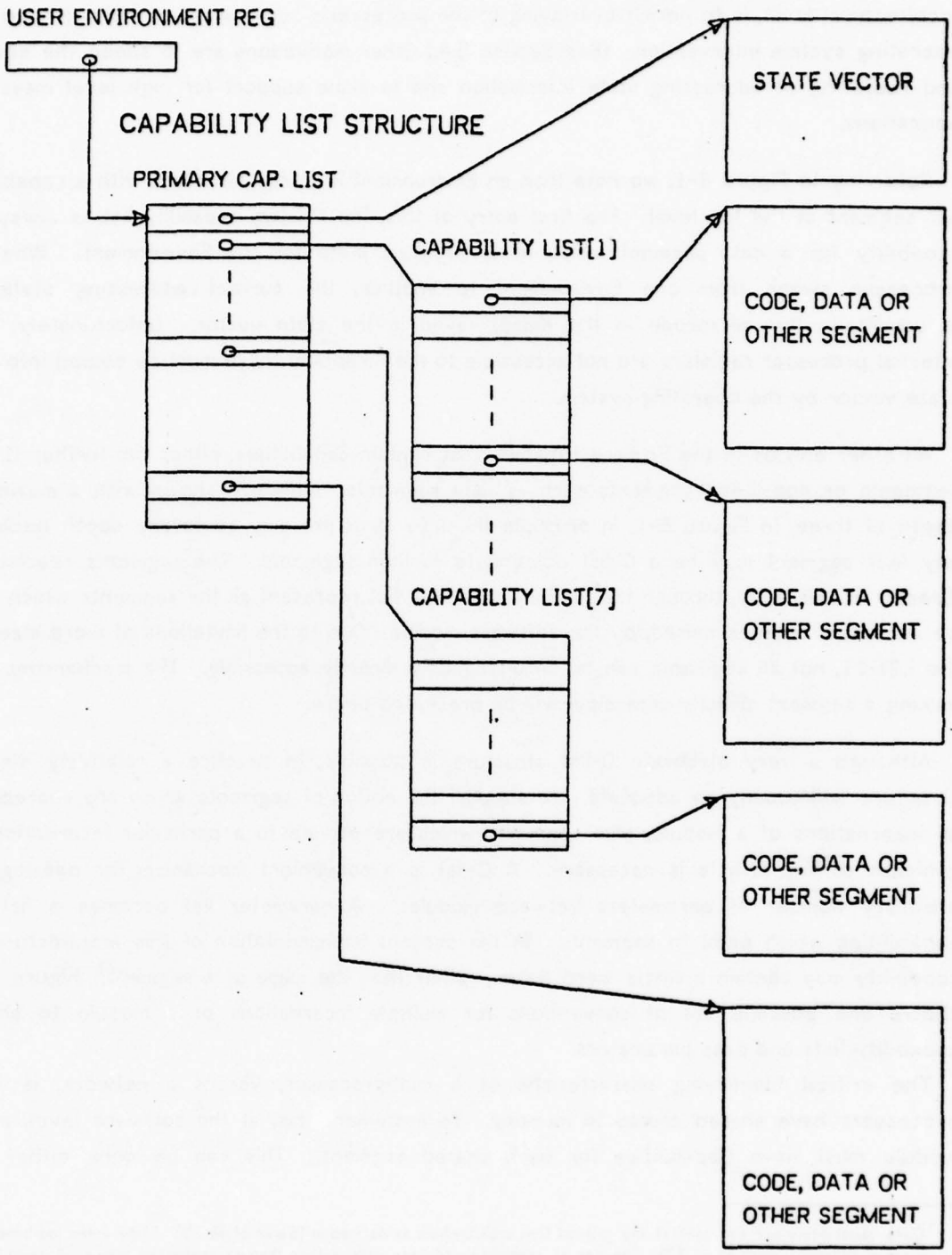


Figure 6-1: The Environment of A Software Module

The primary motivation for defining the representation of a software module at the architectural level, is to permit overlaying of the processor's address space without need for operating system intervention. (See Section 5.4.) Other motivations are to speed the saving and restoring of addressing state information and to allow support for high level message operations.

Referring to Figure 6-1, we note than an Environment is a tree structure with a capability list segment at the top level. The first entry of this, the Primary capability list, is always a capability for a data segment which holds program state for the Environment. When a processor swaps from one Environment to another, the current addressing state is automatically (by microcode in the Kmap) saved in the state vector. Unfortunately, the internal processor registers are not accessible to the Kmap and these must be copied into the state vector by the operating system.

All other entries in the Primary capability list contain capabilities, either for further C-list segments or non-C-list segments such as data segments. Although shown with a maximum depth of three, in Figure 6-1, in principle the tree structure may go to any depth because any leaf segment may be a C-list pointing to further segments. The segments reachable, directly or indirectly, through the Primary capability list represent all the segments which can be accessed, or even named, by the software module. Due to the limitations of word size on the LSI-11, not all segments can be simultaneously directly accessible. The mechanisms for making a segment directly accessible will be presented below.

Although a very elaborate C-list structure is possible, in practice a relatively simple structure will usually be adequate. To support the notion of segments which are shared by all incarnations of a module, plus segments which are private to a particular incarnation, a minimum of two C-lists is necessary. A C-list is a convenient mechanism for passing an arbitrary number of parameters between modules. A parameter list becomes a list of capabilities which point to segments. In the present implementation of this architecture, a capability may contain a single word datum, rather than the name of a segment[1]. Figure 6-2 shows one possible set of conventions for multiple incarnations of a module to share capability lists and pass parameters.

The critical identifying characteristic of a multiprocessor, versus a network, is that processors have shared access to memory. To implement this, at the software level, each module must have Capabilities for each shared segment. This can be done either by

---

[1]Data capabilities are not part of the original Cm• architecture as defined in [Swan et al, 75]. They were implemented to support StarOS [Jones et al, 77]. The use of capabilities to pass data, rather than as protected names of objects, is a deviation from the conceptual consistency of the addressing structure. An alternative, for passing small amounts of data, is to use a pool of segments 10 or 20 words long.

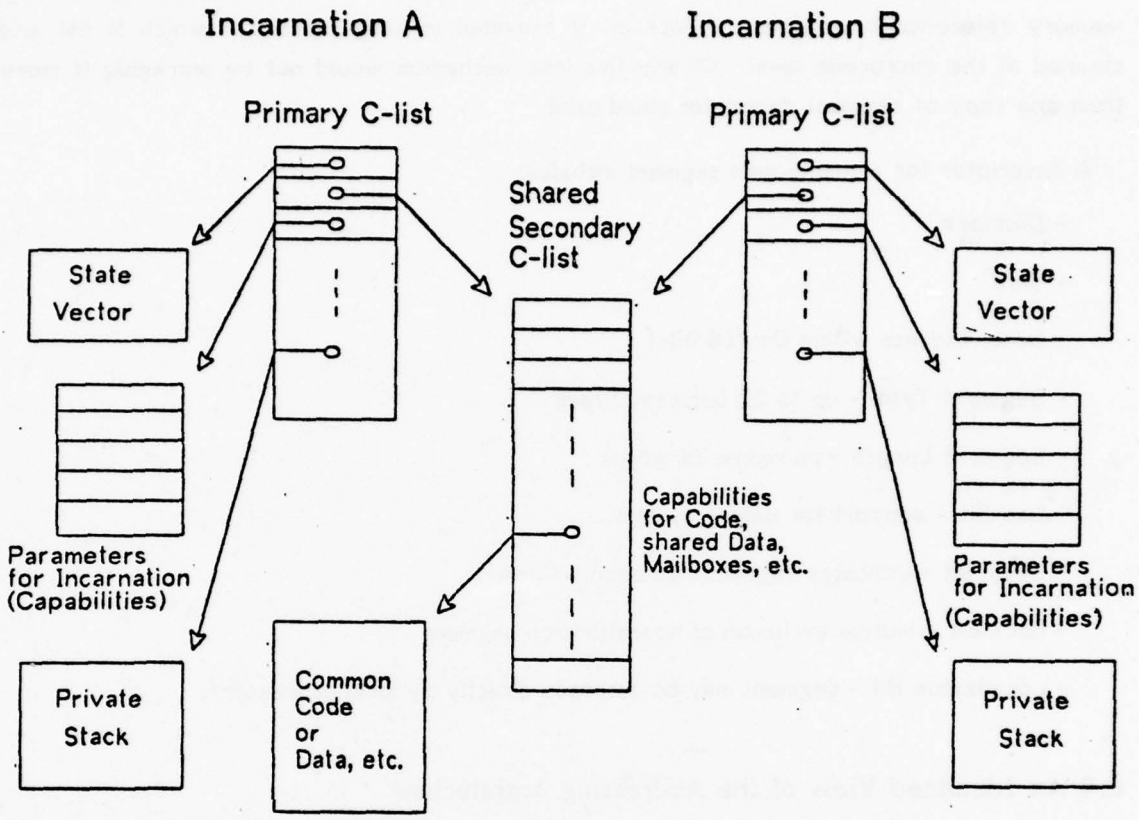


Figure 6-2: Two Module Incarnations

providing separate copies of each capability or, sometimes more conveniently, by sharing a capability list.

6.3.3 Segment Descriptors

A **segment descriptor** is a primitive data type, as is a capability, which is understood and supported by the Cm* architecture. For each segment in the system there is exactly one segment descriptor which defines the type and physical location of the segment. The prime motivation for the insistence on a single (logical) copy of a descriptor is to enable indivisible updates, for example when moving a segment from one physical memory to another. This is discussed at length in Chapter 5. The typed segment mechanism in Cm* depends heavily on

being able to lock a segment for a short time, with very low overhead, during multiple memory reference operations. A lock bit is provided in each descriptor which is set and cleared at the microcode level. Clearly this lock mechanism would not be workable if more than one copy of segment descriptor could exist.

A descriptor for a simple data segment includes:

- Cluster#
- Cm#
- Base Address within Cm (18 bits)
- Segment Type - up to 32 Segment Types
- Segment Length - maximum 2K words
- Use Bit - support for paging system
- Dirty Bit - indicates segment has been written
- Lock Bit - mutual exclusion of operations on segment
- Localizable Bit - segment may be accessed directly by local processor[1].

6.4 An Idealized View of the Addressing Architecture

In the previous section we reviewed the primitives that a programmer needs to understand in order to use Cm*. For the most part, discussion of the implementation of the addressing architecture has been avoided. In this section we will complete this idealized, or conceptual, view of the architecture. We will delay, for subsequent sections, peering beneath this shiny exterior to see the implementation on the physical structure of Cm* using microcomputers.

To summarize the idealized architecture:

- The single, basic primitive object in the system is a **segment**. Segments are abstract data types with a small number of operations defined on them. One kind of segment corresponds to a B5500 style variable length, unpaged segment.
- Segments can only be referenced via a **capability**, which both names a segment and specifies which operations on the segment (and capability) are permitted.
- Segments are defined by a **segment descriptor**, which gives its type, physical location, etc. There is a one to one correspondence between segment descriptors and segments.

---

[1] See Section 6.7.4.

- Capabilities may be combined in a linear list called a **capability list segment**. These segments may be combined to form a tree structure of Capabilities, called an **Environment**.

- An **Environment** represents the entire state of an executing software module. Special operations are provided to facilitate the saving and restoring of this state.

- The **capability lists** of an Environment define the translation of Execution Environment Names[1] into System Names[2]

- Segment descriptors may also be combined into linear lists, called **descriptor segments**. Descriptor segments are combined to form the **descriptor directory**. This is implemented as a distributed structure, see Section 6.7.3.

- The **descriptor directory** defines the translation from the system Name Space to the physical memory address space.

Figure 6-3 shows the entire, idealized, addressing structure. The circled numbers on the figure correspond to the following steps:

1. On the left is an address generated within the Execution Environment, i.e., an address from the processor which is completely under the control of the programmer.

2. The high order part of the address is used to select a capability from the C-list structure of the Environment.

3. The capability specifies a segment. This name is used to retrieve a segment descriptor from the descriptor directory.

4. The rights field in the capability is compared with the operation code from the processor to ensure that the operation is permitted.

5. The segment type, in the segment descriptor, indicates that the segment is a normal data segment.

6. The operation code, combined with the segment type, indicates a simple read operation is requested.

7. The limit field in the segment descriptor is compared with the offset value from the low order part of the original address. If the offset exceeded the limit, a segment bounds overflow error would be flagged.

8. The segment descriptor directly specifies the Cm holding the segment.

9. The base address from the segment descriptor is added to the offset to give a

---

[1] Addresses directly generated by a processor during execution of a program. See Chapter 4.
[2] Names of objects within the System Name Space. See Chapter 4.

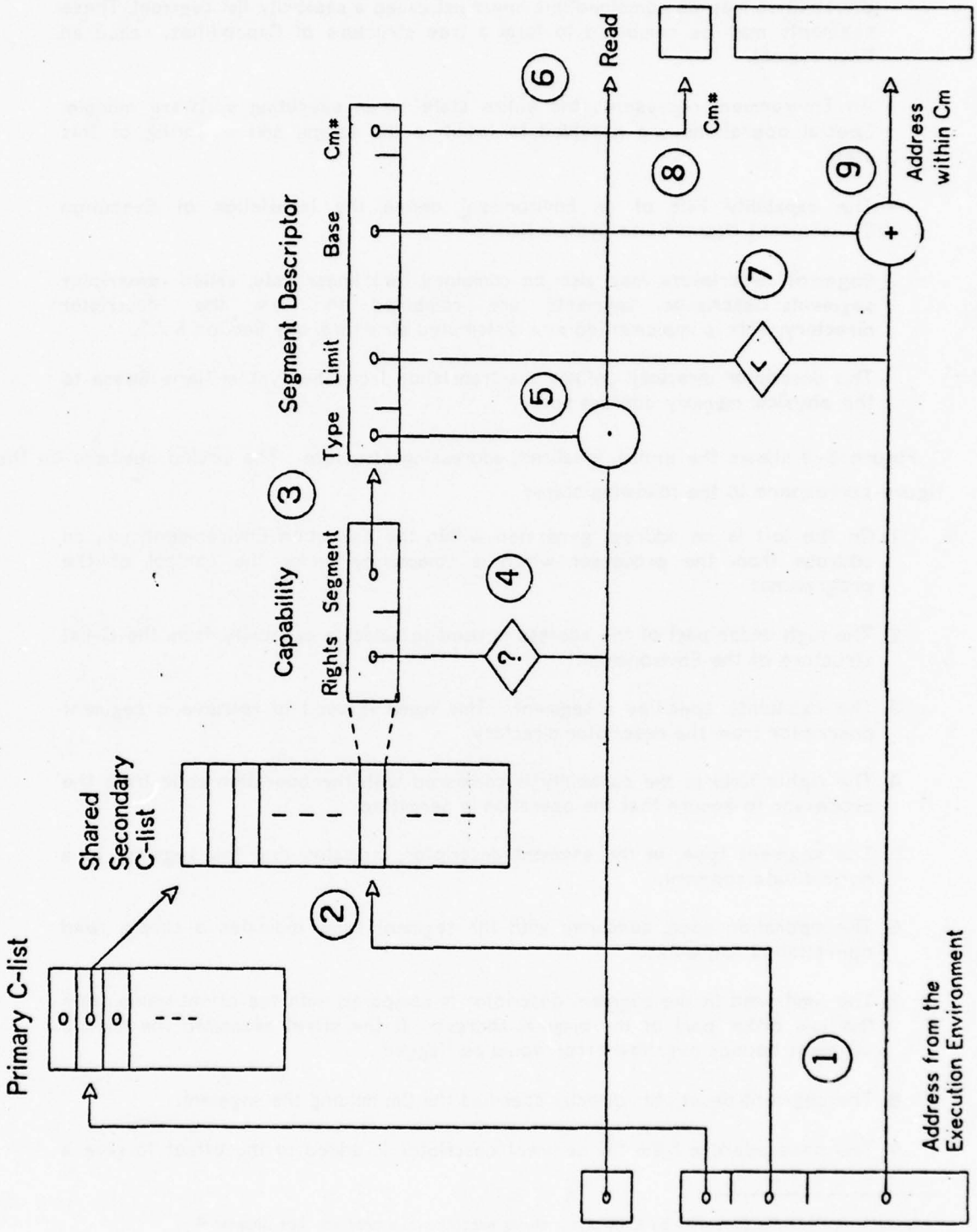


Figure 6-3: The Conceptual Addressing Architecture

physical address within the designated Cm.

6.5 Approach to Implementing the Addressing Architecture

In the discussion of the operation and implementation of the Cm* addressing architecture, the reader should keep in mind the following:

1. The instruction set and addressing modes of the LSI-11 have not been changed in any way.

2. Most of the power of the addressing structure comes from the microcoding of the address translation process in the Kmap. The Kmap has substantial, fast, internal data storage for mapping tables and "pseudo registers". Pseudo registers are registers which appear in the address space of individual processors, but are actually implemented within the Kmap.

6.6 The Page 15 Mechanism

We will see below that the immediate address space of the LSI-11 processors is subdivided into 16 divisions. Each subdivision, or page, can be used to access a distinct segment. The "top" subdivision, page 15, is reserved for communication with the Kmap. References to this page are always mapped (except during startup) and are tagged when passed to Kmap. The programmer is presented with a number of pseudo registers which are directly accessible as memory locations; however, references to these locations are interpreted in Kmap microcode. Thus a reference, read or write, to any location within page 15 can invoke an arbitrary microcode sequence. The address offset within the page, the operation type (read/write), and the data value (for a write operation) are used as parameters. Inspection of the format for transactions over the Map Bus, Figure 4-5, shows that two bits, Space and God, from the current external Processor Status Word are also passed to the Kmap microcode. These bits are used to distinguish User/Kernel mode and highly privileged programs, respectively. We will see that use of page 15 is central to the high level of support for operating system primitives provided in Cm*.

6.6.1 The Page 15 Registers

Figure 6-4 shows the Kmap communication pseudo registers provided for programs operating in the kernel address space of a Cm. These registers are a superset of those provided for the user address space, Figure 6-5. The registers may be considered in several distinct groups:

1. Startup and Initialization. The Directory Base register defines the physical start address of the descriptor directory structure. This is one of the very few

occasions when a direct physical address is used[1]. This register is set once, and by only one Cm, at system creation. The **Kernel Environment** Register is also only set during system startup or when initializing a processor. This register contains the segment name (a 16 bit integer) of the Primary C-list for the instantiation of the kernel which will execute on this processor. This is the only occasion when it is necessary to specify a System Name directly, rather than via a capability. The addressing structure is not defined until these registers are set. Therefore the startup sequence must execute with local code and only references to page 15 are passed to the Kmap. For the sake of protection these registers can only be set while in God mode (XPSW<14> = 0) and are only visible in the kernel space.

2. The **Control Stack** register specifies the stack segment for saving and restoring the PC and PSW during interrupt and trap sequences (see Section 4.7). The **Interrupt Vector** register specifies the segment used for the transfer vector during interrupts and traps. These segments are specified via C-list indices, which select a capability from the Kernel Environment. Normally the same control stack and interrupt vector segments can be used for all users. This scheme allows the flexibility of each User Environment specifying its own interrupt vector (which must be vetted by the kernel) and selectively servicing its own interrupts and traps.

3. The **User Environment** register selects a segment representing the current User Environment. The Kernel Environment must include a capability for the User Environment. (See Figure 6-6.) Writing the C-list index of a new User Environment into the register causes the addressing state of the current user to be saved, by the Kmap, in the state vector of that User Environment. The addressing state of the new user is loaded from its state vector. Thus in principle, a context swap is achieved by a single instruction within the kernel. Unfortunately, the Kmap does not have access to state within the LSI-11; this must be saved and restored in a conventional way.

4. The **Error Mask** and **Error Status** registers allow the response to various exception conditions to be specified. Each bit in the Error Status register corresponds to a class of detected error. If the corresponding bit in the Error Mask is set when an error occurs, then an error trap will be forced; otherwise the program will explicitly test for errors. Only the low order byte of the user Error Mask is under direct user control. The kernel space has its own error registers plus access to the user error registers. The **Error Code** register gives a more detailed description of the error detected. Complete information about the reference which caused the error is also given. This includes the processor generated address; or type, system address and physical address[1].

5. The **Window** Registers are used for expansion of the Execution Environment name space. This is described below.

---

[1]The only other occasion is when specifying a base address for a segment when creating a segment descriptor.

[1]This complete information block is not implemented in the current microcode.

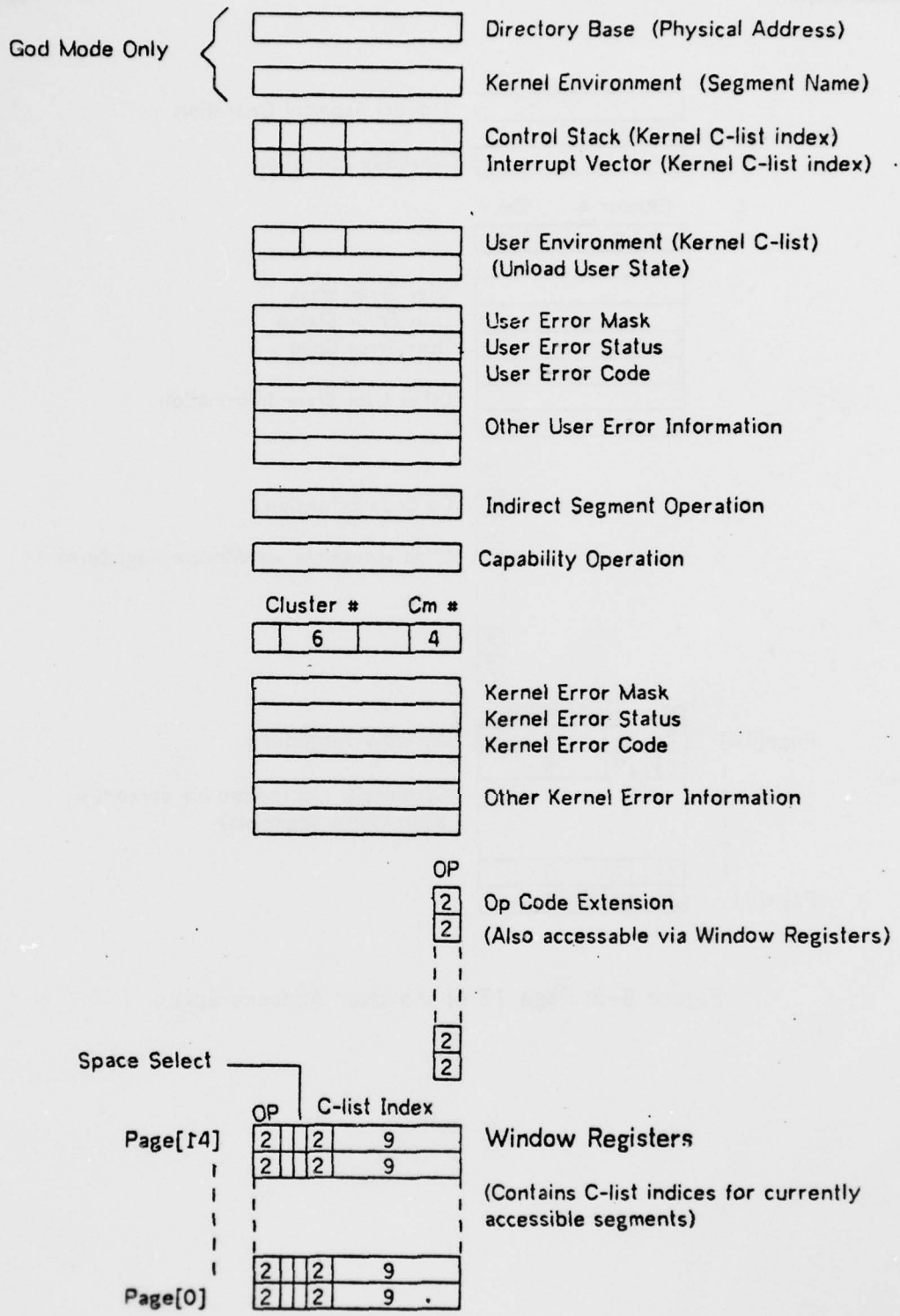


Figure 6-4: Page 15 of the Kernel Address Space

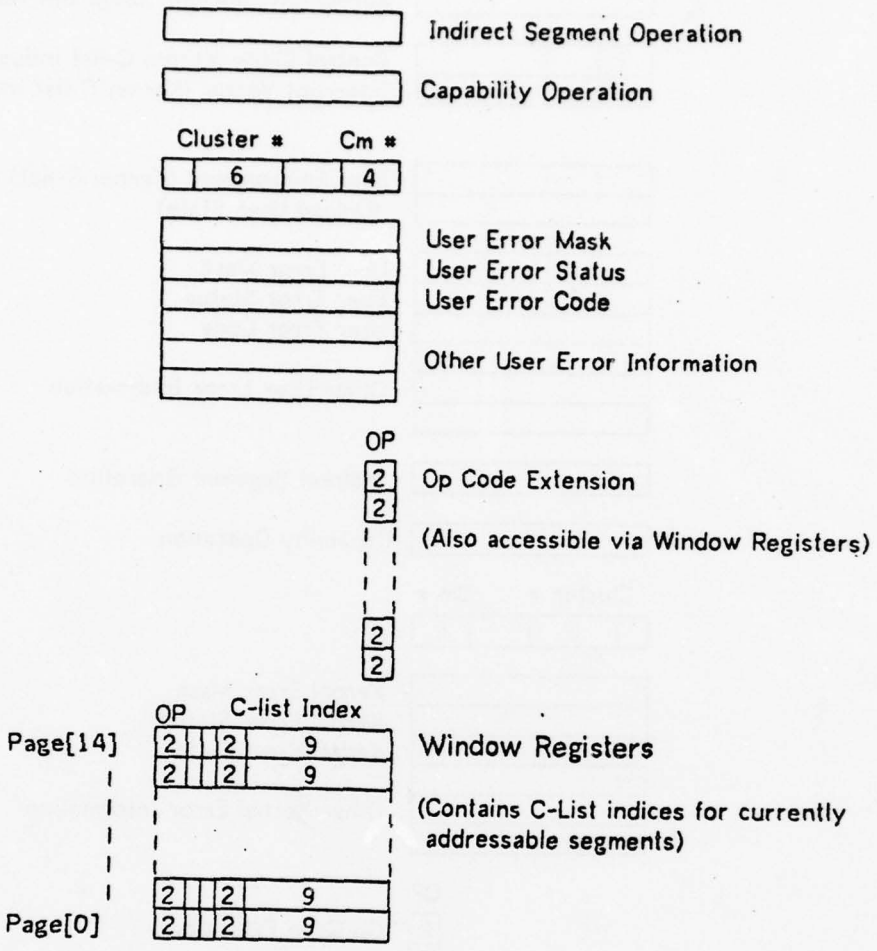


Figure 6-5: Page 15 of the User Address Space

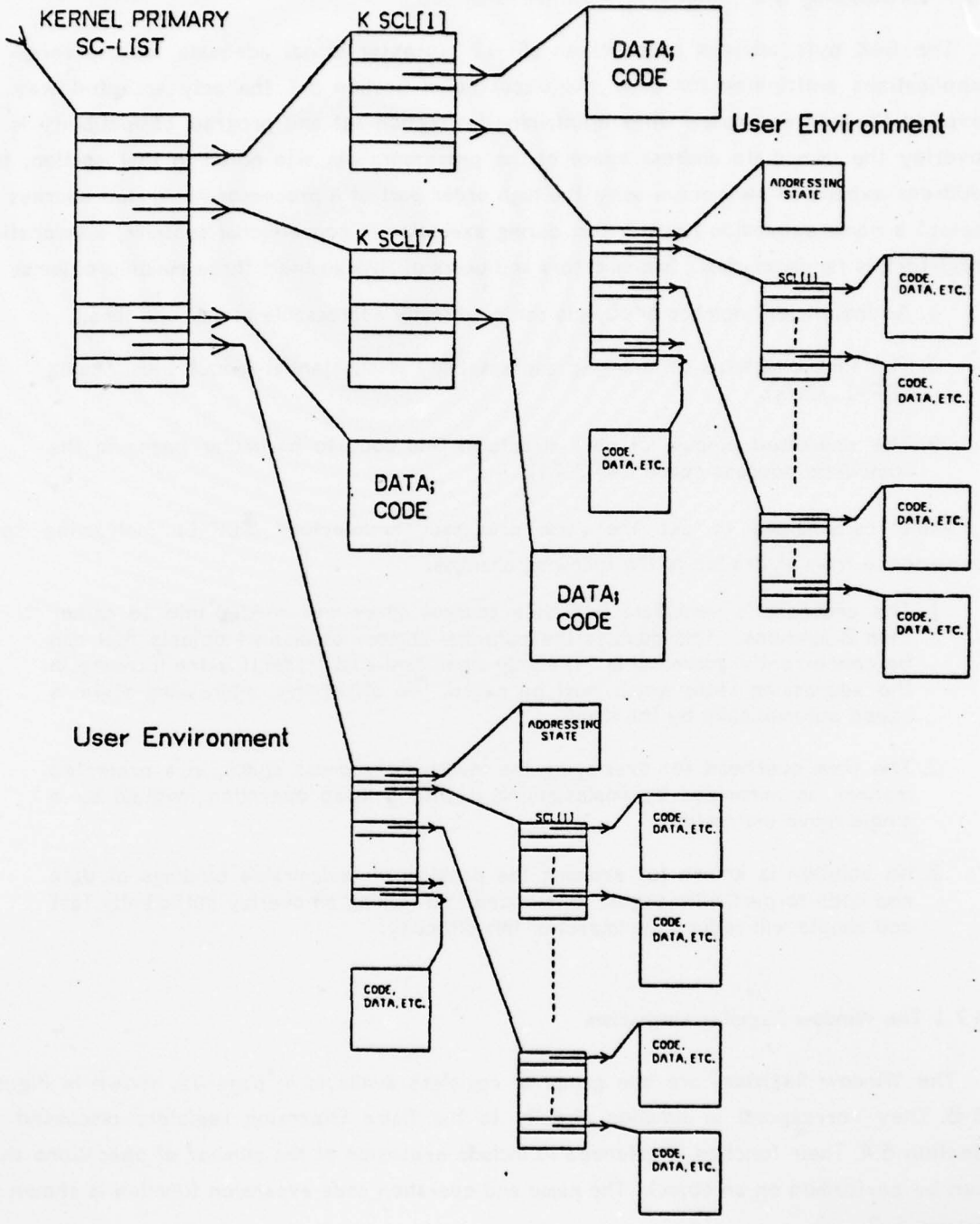


Figure 6-6: A Kernel Environment

6.7 Expanding the Processor's Address Space

The 64K byte address space of an LSI-11 processor is not adequate for the range of applications anticipated for Cm*. As discussed in Section 5.4, the only accepted way to expand the address space while maintaining instruction set and program compatibility is to overlay the immediate address space of the processor. As also noted in that section, the address expansion mechanism using the high order part of a processor generated address to select a name expansion register (or, during execution in conventional systems, a relocation register), is far from ideal. Implementors and users of Hydra report three major problems:

1. An insufficient number of objects can be directly addressable at any one time.
2. The time overhead of changing addressability is substantial (about 1 us. using PDP-11/20's).
3. The unwanted binding of data structures and code to particular pages in the immediate address space (see 5.4.1).

While constrained to use the same processor architecture (PDP-11 instruction set), experience from Hydra led to the following changes:

1. The processor's immediate 64K byte address space was divided into 16 rather than 8 sections. This doubles the potential number of distinct objects that can be concurrently addressable. The only significant cost tradeoff is the increase in the addressing state which must be saved. To offset this, addressing state is saved automatically by the Kmap.
2. The time overhead for overlaying the immediate address space, in a protected manner, is minimized by implementing it with a Kmap operation invoked by a single move instruction.
3. No solution is known for avoiding the problem of undesirable bindings of data and code to particular pages. It is hoped that making an overlay sufficiently fast and simple will reduce the impact of this difficulty.

6.7.1 The Window Register Mechanism

The **Window Registers** are one group of registers available in page 15, shown in Figure 6-5. They correspond in function directly to the Name Expansion registers discussed in Section 5.4. Their function is extended to include expansion of the number of operations that can be performed on an object. The name and operation code expansion function is shown in Figure 6-7.

Each window register entry is defined as follows:

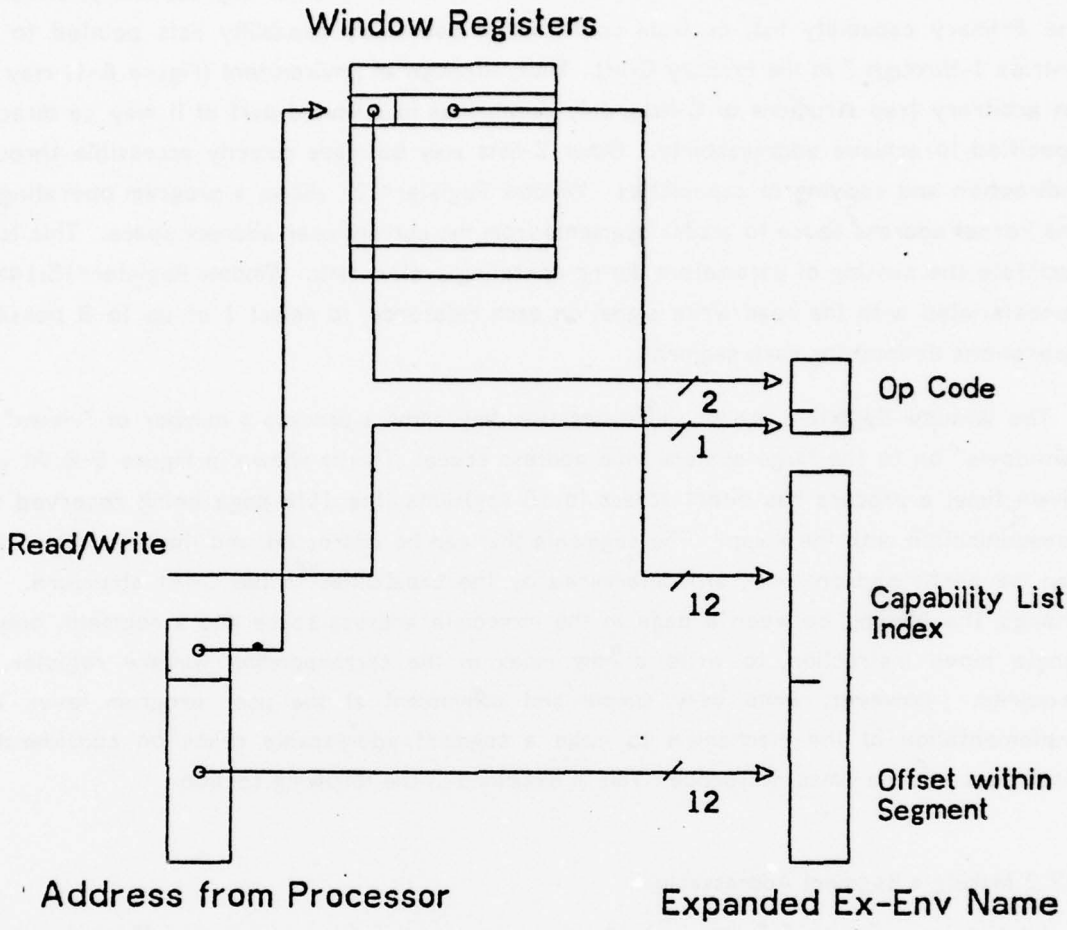


Figure 6-7: Address and Operation Code Expansion with Window Registers

Window Register<15:0>

<15:14>	Op Code Extension
<13>	Reserved
<12>	Use current User Environment (kernel only)
<11:9>	Select Primary C-list (0) or Sec. C-lists (1-7)
<8:0>	Select capability within C-list

We see that Window Register<11:0> allows the direct selection of any capability, either in the Primary capability list, or from up to seven Secondary capability lists pointed to by entries 1 through 7 in the Primary C-list. Thus, although an Environment (Figure 6-1) may be an arbitrary tree structure of C-lists, only capabilities in a limited part of it may be directly specified to achieve addressability. Other C-lists may be made directly accessible through indirection and copying of capabilities. Window Register<12> allows a program operating in the kernel address space to access segments from the current user address space. This is to facilitate the passing of parameters during operating system calls. Window Register<15:14> is concatenated with the read/write signal on each reference, to select 1 of up to 8 possible operations defined for each segment.

The Window Registers are so called because they allow a process a number of "views" or "windows" on to the large system wide address space. This is shown in Figure 6-8. At any given time, a process has direct access to 15 segments (the 16th page being reserved for communication with the Kmap). The segments that can be addressed, and the operations that can be performed on them, are determined by the capabilities in the C-list structure. To change the binding between a page in the immediate address space and a segment, only a single move instruction, to write a new index in the corresponding window register, is required. However, while very simple and convenient at the user program level, the implementation of the mechanism to make a segment addressable relies on considerable intelligence in the Kmap microcode. This is examined in the following section.

6.7.2 Making a Segment Addressable

As shown in Figure 6-3, conceptually a reference requires a minimum of three levels of indirection. If a capability from a secondary C-list is used, or the segment descriptor cannot be accessed directly, then further levels of indirection are required. If each level of indirection required access to data structures in main memory, then a reference would be extremely slow. Further, as described in Chapters 4, the effective use of Cm* depends on ensuring that most references are to local memory and do not require Kmap assistance. We will see that the addressing architecture can be implemented with very little time overhead.

Following from Section 5.2.5, we know that any sequence of mappings can be represented by a single composite mapping, provided that the composite mapping function is updated whenever any of the individual mappings is changed. This is the approach taken to allow segments, contained within the local memory of a processor, to be accessed directly by the processor. The single composite mapping function is performed by the relocation registers in the Slocal.

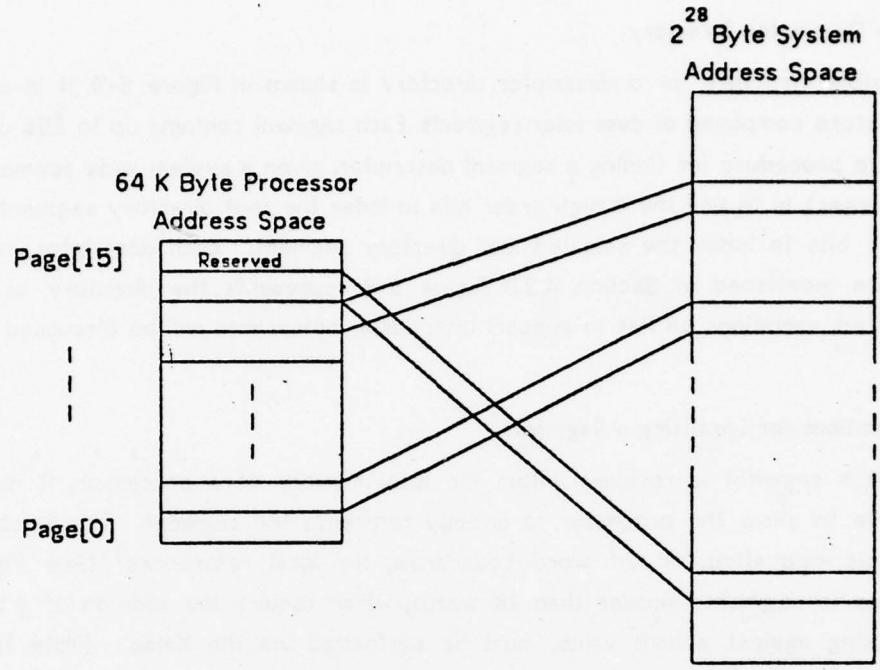


Figure 6-8: Windows from the Processor's Immediate Address Space

When a program requests, by writing a Window Register, that a segment be made addressable the Kmap performs a full elaboration of the address mapping path to find the segment descriptor. If the segment can be accessed locally (see Section 6.7.4), then the base address of the segment (which must be a 2K word, data or code segment) is placed in the relocation register corresponding to the Window Register. Thus references to that segment can be performed without any Kmap support and without address mapping overhead[1].

If the segment cannot be accessed locally, the Kmap sets the Slocal relocation register so that all references to that page are passed to the Kmap. (The Kmap sets the "map" bit in the relocation register, Figure 2-8.) The Kmap also sets its internal mapping tables so that references to the corresponding page are efficiently mapped to the appropriate segment.

---

[1]The LSI-11 processor is synchronous using a four phase clock with a period of 380 ns. Timings are such that the delay of the relocation registers in the Slocal (about 40 ns) can be inserted in the address path of the processor without altering the clock speed. However, with many processor designs there would be a performance penalty for adding relocation.

The mechanism to do this is discussed in more detail below.

6.7.3 The Descriptor Directory

A possible structure for a descriptor directory is shown in Figure 6-9. It is a two level tree structure composed of **descriptor segments**. Each segment contains up to 256 descriptors. The lookup procedure for finding a segment descriptor, given a system wide segment name (a 16 bit integer), is to use the 8 high order bits to index the root directory segment and the 8 low order bits to index the selected leaf directory segment. Each descriptor includes the information mentioned in Section 6.3.3. Figure 6-9 represents the directory as it is now implemented; variations on this to support intercluster references will be discussed below.

6.7.4 Conditions for Localizing a Segment

Even if a segment is resident within the local memory of a processor, it may not be permissible to allow the processor to directly reference the segment. The Slocal provides only simple relocation, on 2K word boundaries, for local references[1]. (See Figure 2-8.) References to segments smaller than 2K words, which require the addition of a base value and checking against a limit value, must be performed via the Kmap. (Note that simple relocation requires 6 bits of state and no arithmetic operations; the more general base-limit scheme requires 28 bits of state and 2 arithmetic operations.) This restriction to 2K word segments for local access appears to be satisfactory for code and, often, for data. However, while for the sake of performance it is important that the stack be accessed locally (stack references are 10% to 25% of all references) the stack often can be substantially smaller than 2K words. This hardware restriction in the design of the Slocal can lead to internal fragmentation in the stack page of a process.

There are other reasons that a segment may not be accessed locally, although it resides in the memory local to the requesting processor:

1. The segment type may require that accesses be performed only by the Kmap. For example, C-lists, mailboxes and directory segments. These segments include protected data types, such as capabilities and segment descriptors, which must be manipulated as a whole to maintain consistency.

2. The segment may be a normal 2K word data segment, but the operation to be performed may require mutual exclusion during multiple references to the segment. For example, indivisible increment and decrement operations to implement locks. A program invokes such operations by changing the Op Type Extension field in the corresponding Window Register. For each setting of the Op Type Extension, there is a pair of operations possible--invoked by the

[1] The principle reasons for not supporting variable size segments in the Slocal are to limit cost and complexity.

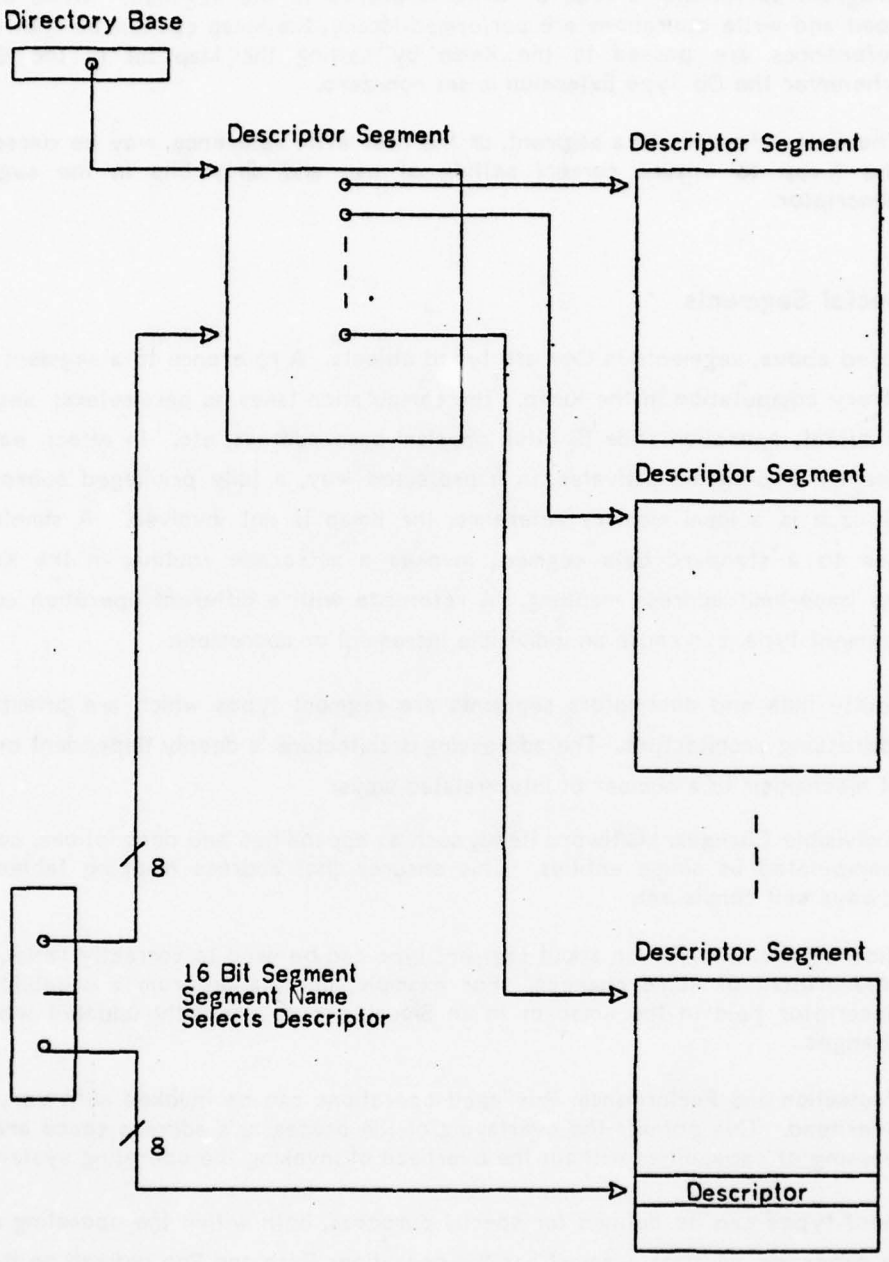


Figure 6-9: The Descriptor Directory for a Single Cluster

program performing a read or write reference to the segment. While simple read and write operations are performed locally, the Kmap can ensure that other references are passed to the Kmap by setting the Map bit in the Slocal whenever the Op Type Extension is set non-zero.

3. The first reference to a segment, or the first write reference, may be passed to the Kmap to ensure correct setting of use and dirty bits in the segment descriptor.

6.8 Special Segments

As noted above, segments in Cm* are typed objects. A reference to a segment can invoke an arbitrary computation in the Kmap. The computation takes as parameters: segment type, address offset, operation code (3 bits), physical base address, etc. In effect, each memory reference by a program activates, in a protected way, a fully privileged subroutine. The simplest case is a local memory reference; the Kmap is not involved. A simple non-local reference to a standard data segment invokes a microcode routine, in the Kmap, which performs base-limit address mapping. A reference with a different operation code, to the same segment type, can cause an indivisible increment or decrement.

Capability lists and descriptors segments are segment types which are primitives in the basic addressing architecture. The addressing architecture is deeply dependent on the typed segment mechanism in a number of inter-related ways:

1. **Indivisible Changes.** Multiword items, such as capabilities and descriptions, can be manipulated as single entities. This ensures that address mapping tables are always self consistent.

2. **Side-effects.** Information about segment type can be used to correctly reflect the side-effects of some changes. For example, information from a capability or descriptor held in the Kmap or in an Slocal, can be correctly updated when it changes.

3. **Protection and Performance.** Privileged operations can be invoked with very low overhead. This permits the overlaying of the processor's address space and the passing of capabilities without the overhead of invoking the operating system.

Segment types can be defined for special purposes, both within the operating system and for user programs. A **stack segment** has the operations Push and Pop defined on it. It is used for saving and restoring processor status during interrupt and trap sequences. (See section (The-PDP11-stack-protection-problem).) Figure 6-10 shows a **queue segment** that might be used for passing data words in a producer-consumer relationship. **Mailbox segments** allow the passing of messages between processes in a general and protected way. Messages are represented as capabilities. The passing of capabilities allows controlled access to large

amounts of data to be passed without the overhead of copying. Capabilities may refer to C-lists, hence large, complex data structures may be passed with a single pointer.

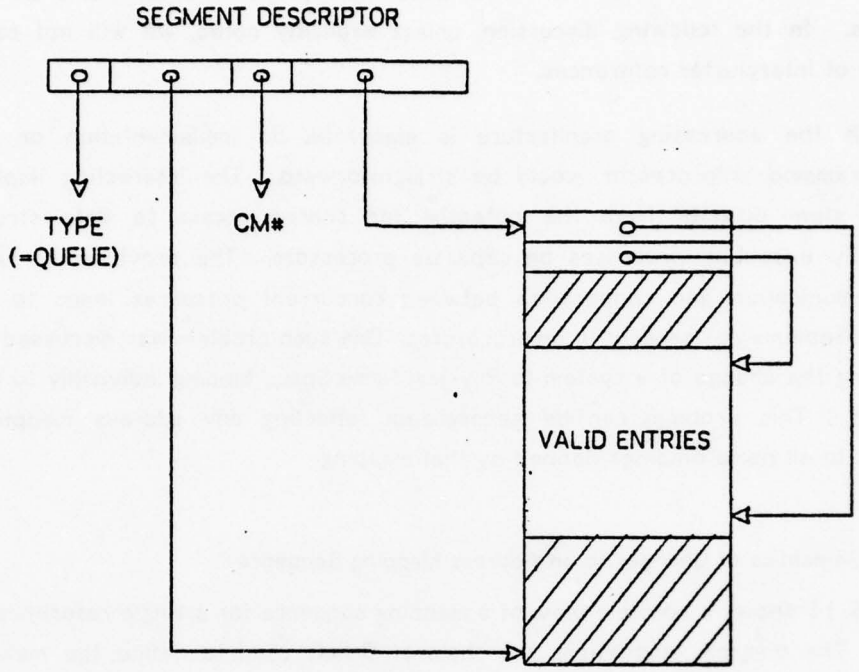


Figure 6-10: A Segment Defined as a Simple FIFO Queue

There are several limitations to the present typed segment mechanism. The computation associated with a segment reference is confined to the Kmap. In future implementations it may be possible for a reference to invoke a software module on an LSI-11 processor. This would be particularly useful for handling exception conditions. It would be desirable if all central transfer operations were invoked in this way.

A severe restriction on the typed segment mechanism is the passing of parameters. For a read-operation, the only parameters from the program are a 2 bit op code extension and a 12 bit offset. The op code extension mechanism is awkward to use. As a result of these difficulties, pseudo registers in page 15 have been used to pass parameter blocks for many operations. This too, is a clumsy mechanism.

6.9 Issues in the Implementation of Within Cluster References

To aid the description of the addressing structure implementation, it is useful to first consider intracluster references before considering the additional problems of intercluster references. In the following discussion, unless explicitly noted, we will not consider the possibility of intercluster references.

Although the addressing architecture is elaborate, its implementation on a suitably microprogrammed uniprocessor would be straightforward. The interesting implementation questions stem directly from the potential for shared access to data structures by concurrently executing processes on separate processors. The provision of facilities for close communication and cooperation between concurrent processes leads to addressing structure problems not present in uniprocessors. One such problem was discussed in Chapter 5; reflecting the change of a system to Physical Name Space binding indivisibly to all users of the object. This problem can be generalized; reflecting any address mapping change, indivisibly, to all name bindings defined by that mapping.

6.9.1 The Semantics of Changes to an Address Mapping Sequence

Figure 6-11 shows a possible view of a mapping sequence for a single reference by a user program. The diagram emphasizes the chain of C-lists used to define the meaning of an address generated by a process. We will assume that the descriptor directory is as described in Section 6.7.3. (To simplify the diagram, the descriptor directory lookups have been omitted from Figure 6-11).

A C-list is merely a particular type of segment; like all segments it can be changed by any process which has an appropriate capability[1]. In this section we will investigate under what circumstances it is reasonable to permit a change to a C-list and what semantics should be attached to changing a capability which is part of an address mapping sequence. We will also consider the affect of other changes to a mapping sequence.

In Figure 6-11 there are three registers, four capabilities and five segment descriptors associated with defining the meaning of a 16 bit address generated by a processor. (Similar structures may be found within many operating systems; in Cm* they are very explicit to allow microcode support for the operating system.) We will first consider the function of the three registers and the affect of a register being changed:

---

[1] segment typing provides an additional level of protection over the capability system. Only operations which preserve the integrity of capabilities (and the reference counts in corresponding descriptors) are defined: e.g., delete, copy, transfer, restrict rights but not increment or write, etc. as defined for data segments.

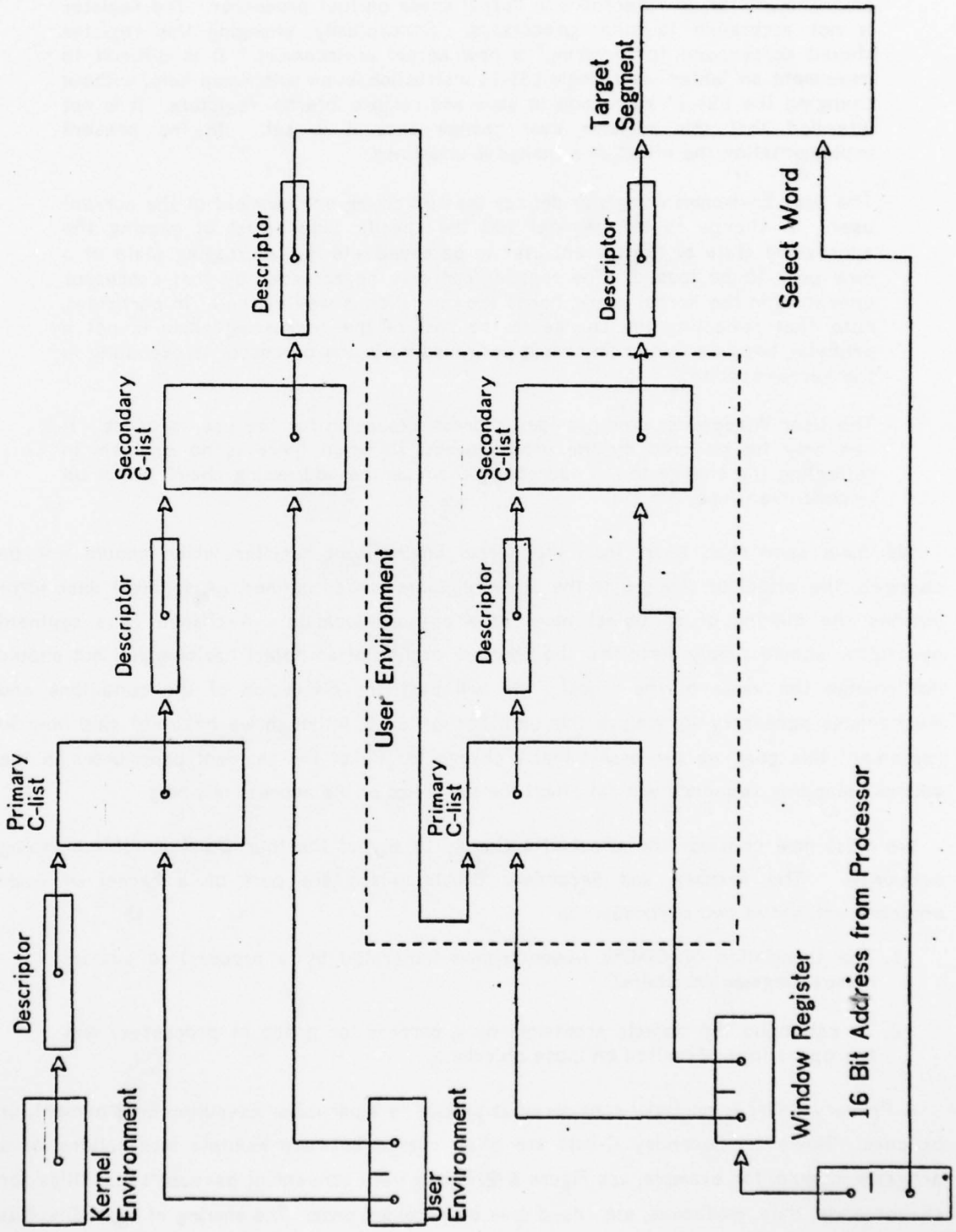


Figure 6-11: An Address Mapping Sequence

- The **Kernel Environment** register (see Section 6.6.1) defines the addressing environment for all references in Kernel space on that processor. The register is not accessible to other processors. Conceptually, changing this register should correspond to "entering" a new kernel environment. It is difficult to implement an "enter" as a single LSI-11 instruction, even with Kmap help, without changing the LSI-11 microcode to save and restore internal registers. It is not intended that this register ever change once it is set. In the present implementation the effect of a change is undefined.

- The **User Environment** register defines the addressing environment of the current user. A change to this register has the specific side effect of causing the addressing state of the current user to be saved and the addressing state of a new user to be loaded. The register can only be accessed by that processor operating in the kernel space, hence the operation is well defined. In particular, note that reflecting the change to the rest of the addressing chain is not a problem, because that chain cannot be in use while the processor is executing in the kernel space.

- The **User Window** register provides address expansion for the user process. It can only be changed by the user process; so again there is no problem in reflecting the change to the addressing chain, as the addressing chain cannot be in concurrent use.

We have seen that, apart from the Kernel Environment register which should not be changed, the effect of changes to the other registers is well defined. A segment descriptor defines the binding of an object name to a physical location. A change to a segment descriptor should simply mean that the physical location of an object has changed but should not change the value of the object. We will postpone discussion of the conditions and mechanisms necessary to achieve this until Section 6.9.2. Although we have not said how to implement this goal, we can assert that a change to any of the segment descriptors in the address mapping sequence will not affect the semantics of the address mapping.

We must now consider the affect of a change to any of the four C-lists in this mapping sequence. The Primary and Secondary C-lists, which are part of a kernel or user environment, serve two purposes:

1. The translation of Ex-Env names (names generated by a process) to system names (segment numbers).

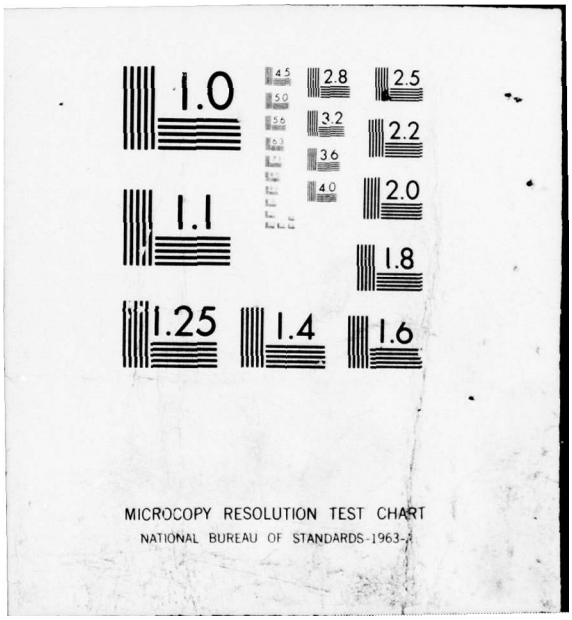
2. To catalogue the objects accessible by a process (or group of processes) and the operations permitted on those objects.

A *Primary C-list is normally considered* as private to a particular Execution Environment, or process. However, Seconday C-lists are often shared between multiple incarnations of a program module, for example, see Figure 6-2. This is very convenient because capabilities for shared code, data, mailboxes, etc. need only be provided once. The sharing of capability lists

AD-A060 494 CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
THE SWITCHING STRUCTURE AND ADDRESSING ARCHITECTURE OF AN EXTEN--ETC(U)
AUG 78 R J SWAN F44620-73-C-0074
UNCLASSIFIED CMU-CS-78-138 NL

3 OF 3
AD
AO 60494

END
DATE
FILMED
1-79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

between processes in a multiprocessor introduces the possibility of a capability being overwritten while it is concurrently being used for addressing.

C-lists, like other segments, are held in main memory. In no practical addressing mechanism would it be reasonable to de-reference through main memory, even once, for each user program reference. In the structure shown, it takes eight main memory references to acquire the four (two word) capabilities involved. Thus any effective implementation of this addressing scheme must involve caching or copying of capabilities. This has a major impact on correctly reflecting changes to C-lists.

6.9.1.1 An Example of Changing a Secondary Capability List Entry

Consider the following simple example: Two Execution Environments, Process A and Process B, have common Secondary C-list (SCL[3], say). Process A has the segment (S_1) referred to by the 23rd capability in the 3rd Secondary C-list (SCL[3]<23>) directly addressable via page 5 of its immediate address space. (I.e., Process A has written 3<23> into Window Register [5].) The following sequence of events occurs:

1. Process A references page 5 and accesses S_1.
2. Process B deletes the capability at SCL[3]<23>.
3. Process B transfers a capability for S_2 to SCL[3]<23>.
4. Process A references page 5 and accesses $S_{??}$.
5. Process A is descheduled, its addressing state is saved and later restored when rescheduled.
6. Process A references page 5 and accesses $S_{??}$.
7. Process A calls an internal library routine which saves the value of Window Register 5, uses Window Register 5 for its own purposes, and then restores the value of Window Register 5.
8. Process A references page 5 and accesses $S_{??}$.

In step 1, Process A references segment S_1 as expected. Which segment is referenced in steps 4, 6 and 8? This simple example raises several questions which are important in the correct and efficient implementation of closely cooperating processes on a multiprocessor.

6.9.1.2 Issues Raised by the Example

Before examining the example in detail, we will mention some of the issues:

A. **Correctness.** Can the situation be handled without compromising the protection

scheme? For example, in step 2, Process B deletes a capability for S_1. If this were the last capability, then segment S_1 might also be deleted. In systems with a limited name space, such as Cm*, the name might be immediately reused for an entirely separate segment (S_3). With some interpretations (or implementations) Process A might reference segment S_3 in step 4. This would be a complete violation of the intent of the protection scheme.

A. *Transparency.* Between steps 4 and 6, Process A is halted and restarted by the operating system. This is an action asynchronous with, and unrelated to, actions of Process A. It can be strongly argued that program behavior should be transparent to such actions and hence that Process A should reference the same segment in steps 4 and 6. Between steps 6 and 8, an internal procedure is called. This is deterministic and directly under the control of the process. Yet, likewise it can be strongly argued that it should be possible to introduce standard debuggers, IO packages, etc. which act transparently to the rest of the program. Calling a routine which saves and correctly restores internal general purpose registers is transparent with respect to those registers. The same should apply to the Window Registers, which are essentially index registers for address expansion. Thus Process A should reference the same segment in steps 6 and 8.

A. **Shared, writable C-lists?** It can be questioned whether it should be permissible to change entries in shared C-lists. The alternative to sharing C-lists is for a copy of the C-list to be made for each incarnation of a process[1]. One motivation for not duplicating C-lists is memory cost and time overhead for module incarnation. In the 50 processor version of Cm*, it will be routine to have 50 or more incarnations of a module. For this, and larger future systems, the memory necessary for duplicated C-lists would be substantial. Even on small systems, the time overhead for copying a C-list when invoking a module would contribute to increasing the grain size for effective module interactions.

A.It does not follow that because C-lists are shared that they should also be writable. It might be proposed that once created, a shared C-list be static. This would inhibit some useful techniques such as acquiring capabilities for IO connections, expanded data space, etc. during execution. Deletion of capabilities is also required for some applications. As deletion is the main cause of problems with shared C-lists it is reasonable to consider schemes where a capability cannot be deleted while in use (i.e., referred to by a Window Register); however, definition and implementation of such a scheme is difficult. (One simple approach, which achieves almost the same effect will be discussed below.)

6.9.1.3 Copy or Cache Capabilities?

Addressing structure implementations for use with shared, writable C-lists fall into essentially two possibilities:

1. Copying. A private copy is made of each capability in an addressing sequence. (The copy must be performed correctly in the sense that it must count as a new

[1] This occurs with segment tables in Multics and with the "core page sets" in Hydra.

capability, for reference count or garbage collection purposes. This prevents premature deletion of the segment.)

2. Caching. The value of the capability is used in such a way that a main memory reference is not necessary, but that any change to the capability will affect the cached copy and main memory copy indivisibly.

The argument for copying is that, once addressability to a segment is established, it is immune to changes to the C-lists by other processes: It is straightforward to implement although incrementing and decrementing reference counts may add significant overhead. It effectively prohibits use of a protection scheme that requires the ability to control the copying of capabilities.

With copying, Process A would refer to the same segment in step 4 as in step 1, segment S_1. Transparency argues that step 6 also refer to S_1. Therefore the private copies of the capabilities must be saved as part of the process state. Transparency also argues that the calling of a procedure in step 7 not affect the segment referenced in step 8. Therefore step 8 must also reference S_1. This is where this implementation breaks down. When re-establishing addressability in step 7 by setting Window Register 5 to [3]<23>, there is no choice but to associate the capability currently in that position in the C-list with that page. The capability is for segment S_2, not S_1. Hence addressability to S_2 is established and transparency is lost.

A caching solution would directly implement the addressing abstraction as defined here and in the previous chapter. All changes to the capability in the C-list at [3]<23> would be indivisibly reflected to all users of the capability. (Concurrent uses of the capability would be resolved into sequential operations on transaction, i.e., user program memory reference, boundaries. Thus in step 4 and subsequent references in steps 6 and 8, Process A would reference S_2.

From a programming abstraction viewpoint, the main advantage of the caching (with indivisible update) implementation is that the addressing architecture behaves in a completely predictable way. A shared C-list behaves like any other shared data structures; updates are reflected immediately. The Window Register mechanism behaves exactly as advertised, i.e., logically the C-lists are indexed for each reference. A disadvantage of this approach is that it is substantially more complex than copying.

6.9.1.4 An Alternative View, and Implementation with Copying

It is possible to view the Window Registers as an explicit C-List which will be manipulated by the process. This view is equivalent to defining the Execution Environment name space, of

a software module, to utilize 16 bit addresses, rather than 21 bit addresses[1]. With this view, a Window Register load (copy) is an alteration of the EE → S mapping[1]. In the author's view, as expressed in Chapter 5 and in the design and implementation of the Cm* architecture, the Window Registers are merely a form of index register, giving access to a 21 bit address space.

With the Window registers viewed as an explicit C-List, it is possible to solve, with copying, the problem of transparency raised above. However, additional mechanisms are required. The library routine called in step 7, would have to save the actual capability in Window Register 5, not just the index. A C-List index is simply an integer and can be saved on stack, stored in a register and manipulated in any way required. Saving a capability is considerably more complex. For protection reasons, the capability cannot be directly accessed. It must be accessed via the Kmap. Each process would have to have one or more C-Lists simply for the purpose of temporarily holding capabilities saved from the Window Registers during nested overlay operations. Allocation of entries in these temporary-storage C-Lists would have to be carefully managed. Where Window Register loads from the standard Primary and Secondary C-Lists would be Capability-Copy operations, Window Register loads from the temporary-storage C-Lists would be transfer operations.

As already noted, the issue of copying or caching Window Register entries would not arise if it were impermissable to delete capabilities in shared C-Lists. The same problem does not occur with the Window Registers as an explicit C-List simply because it would be private to a single process. However, it would seem more reasonable to prohibit delete operations on capabilities which are potentially part of an addressing chain. This would allow a process to work directly with 2^9 capabilities, rather than the 15 which can be held in the Window Registers. (See Section 6.11.)

6.9.1.5 The Present Implementation on a Single Cluster

The current version of the microcode (April 1978) supports only a single cluster Cm* system. It was implemented by John Ousterhout. (Other versions of the microcode implemented by John Ousterhout and Pradeep Sindhu, support a simpler architecture for multicluster use.) It provides an almost complete solution to the problems described in this section. A simplified version of the internal Kmap data structure is shown in Figure 6-12.

Both segment descriptors and capabilities are cached, via microcode managed data structures, in the Kmap. Since all references to C-lists and segment descriptor segments go

---

[1] The 9 bit C-List index plus 12 bit offset gives a 21 bit address.

[1] See Section 5.2.

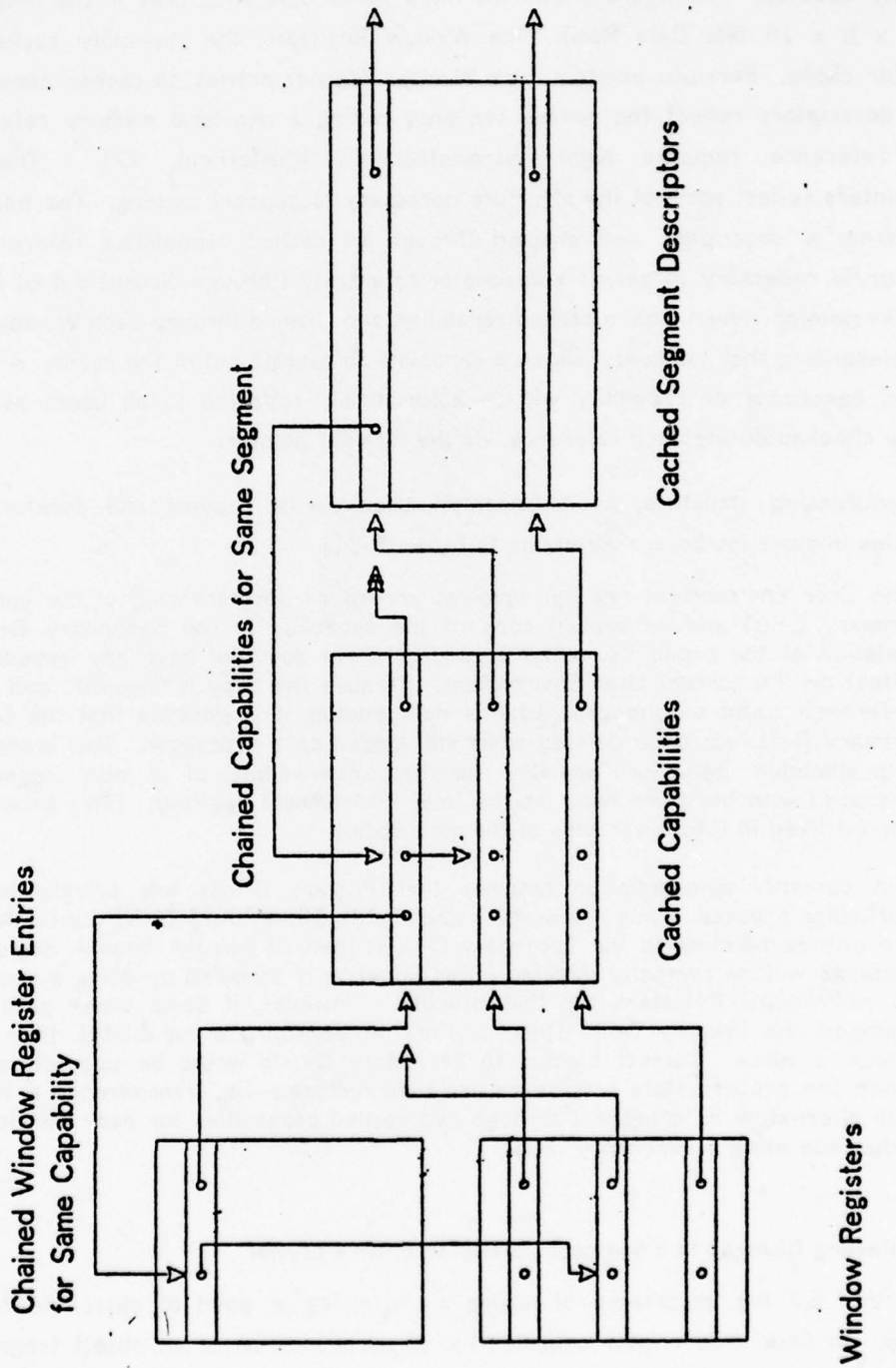


Figure 6-12: Kmap Data Structures for Single Cluster Implementation

via the Kmap, it is possible to ensure that the cached capabilities and descriptors are indivisibly updated. The figure shows the three major data structures in the Kmap (held in the 1K x 5 x 16 bits Data Ram): the Window Registers, the capability cache and the descriptor cache. Forward pointers from Window Register entries, to cached capabilities, to cached descriptors reflect the normal sequence during a non-local memory reference. A simple reference requires eight microinstructions [Ousterhout, 77]. The chained back-pointers reflect some of the structure necessary to support caching. The back-pointer shown from a descriptor, and chained through all cached capabilities referencing that descriptor, is necessary to permit a descriptor to migrate (through disuse) out of the cache. The back- pointer shown from a cached capability, and chained through each Window Register entry referencing that capability, allows a capability to migrate out of the cache. A change to a cached descriptor or capability will be automatically reflected to all users as they are explicitly checked during each reference, via the forward pointers.

The addressing structure, as implemented, uses partial copying and dereferencing of capabilities in some instances. Referring to Figure 6-11:

1. The User Environment register involves an implicit dereferencing of the kernel Primary C-list and an implicit copy of the capability in the Secondary C-list. Deletion of the capability in the Secondary C-list does not have any immediate effect on the current User Environment. Because the copy is "implicit" and the reference count on the descriptor is not updated, it is possible that the User Primary C-list could be deleted while still loaded on a processor. This leads to unpredictable behavior, possibly including overwriting of a new segment allocated with the same name as the User Environment segment. (This problem will be fixed in future versions of the microcode.)

2. The current implementation assumes that Primary C-lists are private to a particular process. Thus if a process updates its own Primary C-list (particularly the entries pointing to the Secondary C-lists) then all current Window Register bindings will be correctly updated. This updating is achieved by doing a search of all Window Registers for that process. However, if some other process changed the Primary C-list (this requires an appropriate capability), then no check is made. Correct binding to Secondary C-lists would be updated only when the process state was saved and then restored--i.e., transparency is lost. The alternative is to indirect through two cached capabilities for each non-local reference using a Secondary C-list.

6.9.2 Reflecting Changes to a Segment Descriptor within a Cluster

In Section 5.7 the importance of having a single, logical point of object definition was examined. In Cm*, a descriptor specifies the physical location of an object (segment). In operation, three actual copies of this information may exist. The permanent, and defining, descriptor is in main memory as an entry in the descriptor directory; a copy may be cached

within the Kmap; a third copy may exist as an entry in the relocation registers of a processor that has the segment locally accessible.

These three copies are updated indivisibly, hence logically there is only a single copy. The segment typing mechanism greatly facilitates this indivisible updating. Any operation on a directory segment which changes a descriptor is done in the following steps:

1. If the descriptor is not in the cache, create a cache entry with the descriptor locked and copy information from the descriptor in main memory. If it is already in the cache, lock it.

2. If the segment is "localized" then "unlocalize" it by changing the Slocal relocation table for the Cm holding the segment.

3. All references to the segment must now all go through the descriptor in the Kmap. Because this descriptor is locked, all references will be blocked. (References to other segments will proceed normally.) The descriptor can now be changed as desired. The data associated with that segment can be moved to a different physical memory. The copy of the descriptor in primary memory must also be changed.

4. When the descriptor is unlocked, references will proceed normally using the new descriptor value. Where possible, the segment will be "localized" automatically after the next reference by a Cm holding the segment[1].

6.9.3 Cost and Performance of Microcoded Operations

The motivation for implementing certain operations in microcode, rather than macrocode, is to achieve higher performance. In part, this is to give improved overall system performance relative to hardware cost. Less tangible, but probably more economically important, is the effect on programmer behavior. The efficient implementation of context swaps, support at the lowest level for capability operations, etc. encourages the use of modern programming methodology. The potential reliability and productivity gains from sound modular decompositions of large software systems will almost inevitably far outweigh the relatively minor hardware cost of supporting suitable primitives.

The additional cost of supporting a capability based addressing architecture on Cm*, over and above the minimum necessary to use the system can be bounded. The fabrication cost of the Kmap is approximately $7,000. The cost of a Cm, with 64K words of memory, is

---

[1] The mechanism for dealing with the un-localizing and localizing of segments in these circumstances is not fully implemented.

approximately $6,000[2]. Thus for a full 14 Cm cluster, the Kmap is about 8% of the total cost. The major difference between implementation of a simple architecture and the more elaborate capability architecture lies in the amount of microcode and data storage used. The implementation of a simple multicluster architecture requires 500 (80 bit) words of microstore [Ousterhout, 77]. The present single cluster capability architecture uses 1500 words. This might double for a full multicluster implementation. Further cost reduction might come from optimizing the Pmap design to suit a particular, simpler architecture. (For example, the general field extraction logic might be eliminated.) The Kbus and Linc both represent designs optimized for cost-performance which would be unaffected by a simplification in the architecture. At best, the cost of the Kmap might be halved if restricted to supporting a simple addressing architecture. Thus an upper bound on the hardware cost of supporting the addressing architecture described here, and others that have been proposed, is 4% of the processor and memory cost for a cluster. This does not include the cost of any I/O, secondary storage, cabinets, power supplies, etc.

The performance of various operations provided by the architecture must be taken relative to performance of the LSI-11 processors. Taking a typical instruction time (two memory references) as 7us we have the operation costs shown in Table(Operation-Costs). (The timings are adapted from [Schwans, 77; Jones, 78].)

The high performance of these operations, relative to an implementation in LSI-11 macrocode, is due mostly to the fact that a trap into trusted kernel code on the processor is not required. Under STAROS, kernel entry and exit, with two parameters, is measured to cost approximately 3000us or 428 average instructions [Schwans, 77]. (One optimized kernel entry operation under HYDRA takes about 300 instructions, a more general kernel entry may be longer.) The Kmap implementation also gains performance because code (and much of the data) is fetched from fast internal memory, extensive field extraction and branching logic is provided, and segment descriptors can be locked and unlocked very cheaply. When the Kmap is forced to make a main memory reference, it is actually slower than for an LSI-11 processor; about 6us.

6.10 Implementing the Addressing Architecture on Multiple Clusters

In a multicluster Cm* system, there is no longer a single Kmap responsible for performing address translation for non-local references and caching capabilities and descriptors. The

---

[2]The quoted cost for the Kmap reflects only CMU fabrication costs while the Cm cost includes a commercial profit margin for the processor and memory. However, the cost of the Kmap is inflated by the expensive fabrication technique used (wire wrap and board charges amount to 37% of the Kmap cost). The commercial profit margin on the Cm memory (priced in quantity at $3,000 for 64K words) must be small because this is a highly competitive market.

Table 6-1: The Cost of Operations in the Architecture

Operation	#Instruction Periods	Actual Period
Local Memory Reference (Same as unmodified LSI-11)	0.5	3.5us
Within Cluster Reference (via cached capability and descriptor)	1.4	9.6us
Load segment (make segment addressable)	3.3	23us
Delete capability (includes decrementing the reference count and possible trap to the O.S.)	12.7	89us
Transfer capability (Reference count unaffected.)	7	70us
Copy capability (Increment Reference count.)	18.1	127us
Indivisible Decrement (Used for synchronization)	5.7	40us

useful extensibility of Cm* rests upon the correct and efficient handling of requests between clusters. A key feature of the addressing architecture is that there is a single, uniform System Name space. In this section we will examine the implementation of a uniform addressing environment, across a distributed structure, without central control.

6.10.1 Design Principles

In Sections 5.8 and 5.9 we discussed, in broad terms, principles for implementing

addressing structures on multiprocessors without a single, central address mapping mechanism. In the context of Cm*, we can discuss those principles in more specific terms:

1. **Single point of object definition and control.** For each segment (object) there is, at all times, exactly one cluster containing its descriptor. The Kmap in that cluster is responsible for controlling access to the segment and translating addresses to the physical address space.

2. **Single point of capability caching.** The dangers of multiple cached copies of writable capabilities are obvious. A natural, single place to cache capabilities would be in the Kmap of the cluster containing the C-list from which the capability is copied. However, if the capability referred to a segment in another cluster, then references to that segment would have to indirect through the cluster containing the C-list. To avoid this performance penalty, **capabilities are cached in the cluster of the segment named by the capability**[1].

3. **Location Anticipation for references to segments.** When referencing a segment, only probabilistic knowledge exists about the name of the cluster actually containing the segment. This information, anticipating the location of the segment, is normally correct. If found to be incorrect, there is a method for finding the correct location and updating the location information.

6.10.2 Basic Intercluster Addressing

A basic primitive for the implementation of the addressing architecture is:

> Given a full description of an operation on an object (segment name, offset, operation code and sometimes data) cause the operation to be carried out.

Programs which must make all references via capabilities, cannot invoke operations at this low level but this is the basic primitive used by Kmaps for the implementation of program level references.

The following mechanisms are necessary to implement the basic intercluster addressing mechanism.

1. **Location Anticipation.** Given a segment name, find the name of the cluster "anticipated" to hold the segment. Several different ways of doing this will be discussed below.

---

[1] This is to some extent a violation of the principle of a single point of object definition. The object; the C-list, exists in one cluster but cached copies of components of that object, capabilities, exist in other clusters. However, for each capability there is exactly one place that it can be cached and there is a straightforward mechanism for causing the cached copy to be indivisibly updated (see below). The situation is similar to the additional copy of (part of) a descriptor which exists in an Slocal when a segment is directly accessible. (See Section 6.9.2.) The principle of a single logical point of object definition is maintained provided any change to the object is reflected indivisibly to all copies of the information.

2. **Physical Routing between Clusters.** Given a cluster name, find the first stage of the path to that cluster. A path stage is defined by selecting one of the two Intercluster buses (connected to that Kmap) and specifying the name of the first (if any) intermediate clusters. In a 64 cluster system a full table for this occupies 64 * 7 = 448 bits. In principle the problem is the same as routing packets through the ARPAnet. However, Cm* by being physically centralized, can accommodate 1000 nodes (clusters) with only a single level of intermediate routing. The complex routing algorithms used in the ARPAnet which dynamically adapt to load and availability do not seem necessary for Cm* like structures. However, a simple scheme to generate alternate routings in the face of error would improve reliability. The issue of physical routing between clusters in Cm* will not be pursued further in this work.

3. **Message Transfer between Adjacent Clusters.** This is simply the transmission of messages between clusters which share a common intercluster bus. (See Section 2.4.2.3 and Chapters 3 and 4.)

4. **Performance of the Operation at the Target Cluster.** When a request reaches the cluster containing the designated segment, the operation is performed as if the request had come from a processor within the cluster.

5. **Correcting and Updating Location Anticipation.** If a request reaches its target cluster but finds that the segment is no longer in that cluster then the location anticipation information at the Source Cluster must be updated and the request retransmitted. This involves both finding the correct information and recording this at the source.

6.10.3 Techniques for Location Anticipation

While maintaining the same architecture presented to programmers, and with the same basic implementation strategy, there exists considerable flexibility in how the probabilistic information about the cluster containing a segment is stored. The location information may be specified in a single, system wide directory structure; a directory partially replicated for each cluster; encoded in the segment name; or, in an auxiliary field within each capability. The schemes are of comparable cost and performance. They vary as to simplicity of implementation, effect of partial system failures, need for garbage collection of names and ease of moving segments between clusters.

6.10.3.1 Scheme 1: Partial Replication of Directories

In this scheme each cluster maintains an independent directory structure for all system names. For segments resident within the cluster the full segment descriptor exists. For segments outside the cluster, the "anticipated" cluster is shown. During system initialization, or by convention, blocks of segment names must be allocated to individual clusters. A possible directory structure is shown in Figure 6-13. Names are initially allocated in blocks of

64. Thus a single word can specify the cluster containing 64 segments leading to memory saving. As segments are moved to other clusters, the compact representation is expanded to show individual segment locations.

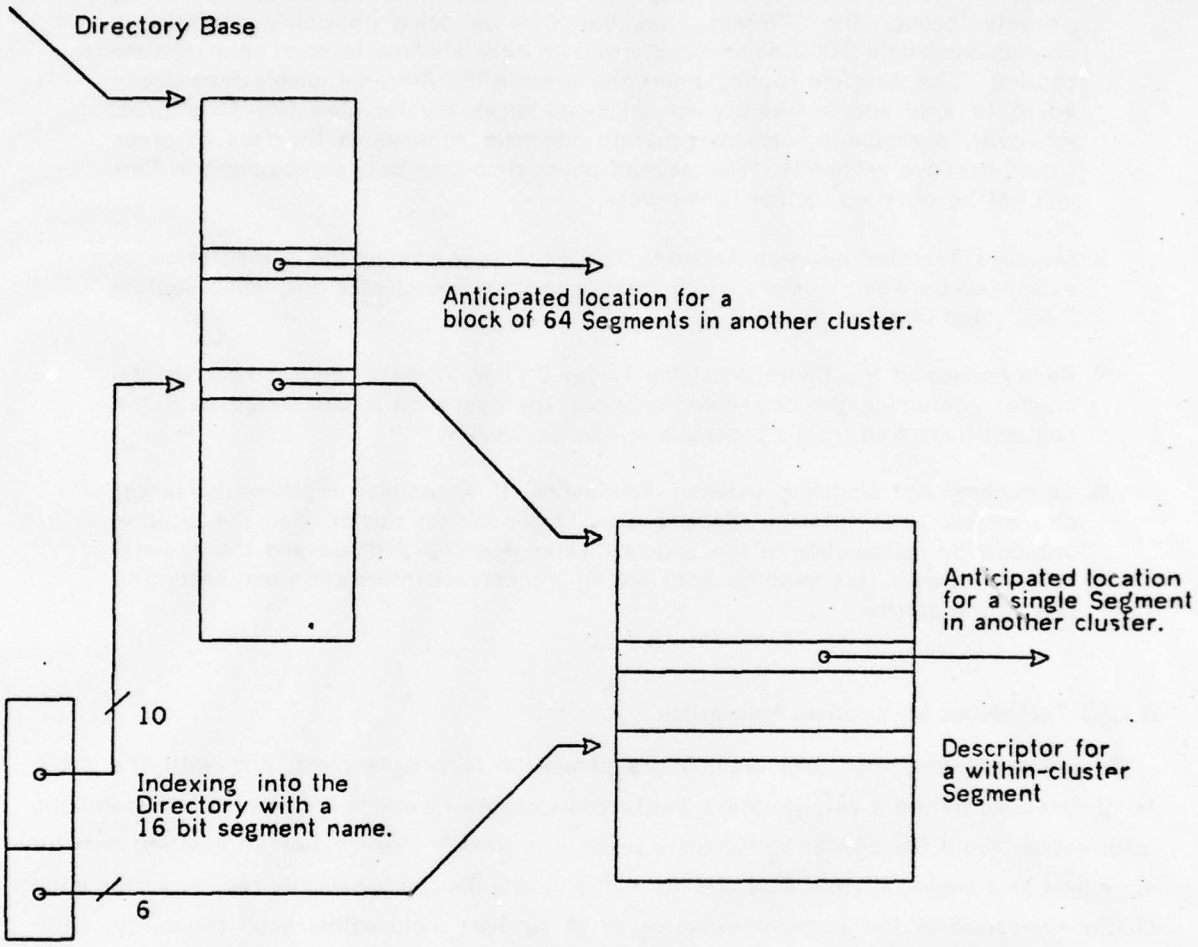


Figure 6-13: Descriptor Directory Scheme with Partial Replication

With this approach, the anticipated locations of all segments, both inside and outside the cluster, are directly available within the cluster. The particular data structure shown is not well suited to manipulation in microcode because a memory block must be allocated when changing from the compact to expanded representation. However, this event may be a rare enough that it can be handled in PDP-11 code. This is the scheme originally proposed for

Cm*.

6.10.3.2 Scheme 2: System Wide Directory Structure

In this approach, there is logically a single directory structure distributed across all the clusters. The structure proposed is an extension of the directory structure currently implemented for a single cluster. (See Section 6.7.3 and Figure 6-8.) The top level of this directory would be replicated in each cluster. As with the scheme above, the name space would be initially partitioned and allocated in blocks for each cluster. The second level directory segments would not be replicated, but would be created within the cluster initially allocated that group of names. This scheme was proposed by John Ousterhout.

In this scheme, even if a segment is moved to another cluster, the descriptor remains in the cluster initially allocated that name. The descriptor is updated, in place, to reflect the new cluster location. This is in contrast to Scheme 1, where a descriptor is always located in the cluster containing the segment.

An advantage of the scheme is that the directory structure can be relatively static and always returns an accurate indication of the current location of a segment. It could be simplified further by using part of a segment name to encode the cluster containing its descriptor. This is only slightly more restrictive than having the location of all descriptors determined at system initialization.

Although static allocation of descriptors to clusters leads to some simplifications, it makes it essentially impossible to partition a cluster out of the system. All descriptors allocated to that cluster must remain there, even though all the corresponding segments have been moved elsewhere. It is also inferior, from a reliability viewpoint, because the demise of either the cluster containing the segment, or the cluster containing the descriptor, can prevent access to a segment.

6.10.3.3 Scheme 3: Encoding of Segment Location in the Segment Name

In this scheme, each cluster has an independent directory. A field in a segment name indicates the cluster containing both the segment and its descriptor. If a segment is moved to another cluster, then it is allocated a new name from the directory in that cluster. The directory entry in the original cluster is changed to give the new segment name.

After a segment has been moved, most capabilities for it will contain the old name. As the capabilities are used they will be automatically updated. The old segment name cannot be re-used until all capabilities have been updated--this requires a periodic sweep of all capabilities to release previously allocated names. This may be part of a general garbage

collection procedure for segments. Alternatively, a reference count scheme can be used both to release the old name and the segment itself.

This scheme appears relatively straightforward to implement and does not have the reliability and reconfigurability restrictions of Scheme 2. It is a little disturbing that an object no longer has a unique name[1]. This scheme was suggested by Bob Chansler and is being considered for use by the STAROS operating system.

6.10.3.4 Scheme 4: Location Anticipation Field in the Capability

A capability could contain both an unique name for an object and its "anticipated" location. This requires more bits in a capability than Scheme 3, but avoids the problems of changing the name of an object and the need to garbage collect names. The location anticipation field may specify more than just the cluster containing the segment, as in the schemes above. The capability could directly specify the anticipated location of the descriptor in the Kmap descriptor cache. Provided this information was usually correct; this might result in a simplification of Kmap microcode without any loss of overall performance. When the location information was correct, performance would be equal to, or faster, than other approaches, because hash-table lookup of the cache would not be required for intercluster references. Incorrect location information could cause a trap to PDP-11 operating system code. This code would maintain data structures to find the correct location and allocate Kmap cache entries. Hence the Kmap microcode could view its descriptor cache as a static table, manipulated only by explicit requests from the operating system. A decision to implement cache management in PDP-11 code (which can be written in a high level language) rather than microcode, depends on the costs of microcode memory and development time and the frequency of cache entry replacements. The macrocode data structures for determining the location of an object may be part of an overall mechanism, required in any operating system, for managing resource allocation for objects.

6.10.4 Caching Capabilities for Intercluster References

Having discussed the basic mechanisms necessary for an intercluster reference, we must now consider references at the program level. All references by a program are via one or more intermediate capabilities. When a segment is made addressable, the sequence of mapping tables is dereferenced to find the identity of the capability actually naming the segment to be referenced. (See Section 6.9.1.5.) This capability is particularly significant for

---

[1] A test for equality between capabilities now becomes quite complex. Both capabilities must be dereferenced to update the current segment name. The descriptor for one segment must be locked while checking the second name to guard against name changes during the comparison procedure.

two reasons:

1. It specifies which operations are permitted on the addressed segment.

2. This capability may be from a shared, Secondary C-list which could be overwritten at any time, by any process sharing that C-list.

The possibility of the capability being overwritten introduces considerable complexity in the implementation. This has been discussed, for a single cluster implementation, in Section 6.9.1. To ensure that changes to the capability are reflected indivisibly to all users of the capability, it is necessary to explicitly check the capability during each reference. A further complication arises because the capability may come from a C-list in a different cluster from the segment named by the capability. As noted in Section 6.10.1, for performance reasons it is important to cache the capability within the cluster containing the designated segment[1].

In Section 6.9.1.5, we saw that in the Kmap data structures a Window Register entry pointed to a cached capability. The capability in turn named the segment which was addressable via that page in the process' immediate address space. If the addressable segment is in another cluster, then the Window Register entry will contain the identity of the capability (segment name and offset) and the anticipated location of the segment. When a reference is made, it is the capability ID which is transmitted to the target cluster, rather than the segment name as in a basic intercluster reference.

When the request reaches the target cluster it will hash into its cache to find the designated capability. If the capability is present, then the operation will proceed like a within cluster reference. If the capability is not present, then the target cluster must use the C-list name and offset to obtain a copy of the capability. If the segment named by the capability is not present in the target cluster, then a mechanism will be invoked to update the location information at the source cluster.

6.10.4.1 Changing a Cached Capability

The steps necessary to delete a potentially cached, capability are as follows:

1. The delete request appears as an operation on a C-list, which must be sent to the cluster containing the C-list segment.

2. The Kmap in the C-list cluster locks the descriptor for the C-list to prevent any possible references to the capability during the delete operation.

---

[1] If the capability were cached in the cluster containing its C-list then all references would have to indirect via that segment. If the capability were cached at the cluster of the process using the capability, then multiple cached copies would result if the capability was used from more than one cluster. This would make it difficult, or impossible, to correctly reflect changes to the capability.

3. The Kmap references the designated capability. If the capability is for a segment within this cluster it decrements the reference count on the segment, deletes a possible entry in the Kmap's capability cache, deletes the capability, unlocks the C-list segment descriptor and returns an acknowledgement.

4. If the capability is for a segment in another cluster, then the name of that cluster is found. A request to delete the designated capability from that cluster's cache is sent.

5. When an acknowledgement is received, a request is sent to the same cluster to decrement the reference count for that segment.

6. When an acknowledgement is received, the capability is deleted, the descriptor unlocked, and an acknowledgement is sent.

6.11 A Review of the Implementation

In examining the implementation of the addressing architecture, we have seen the influence of two factors not present in uniprocessor systems: the physical distribution of the structure without central control and the potential for parallel, asynchronous changes to address mapping tables. These two factors interact to have a substantial effect on the design of the Kmap microcode which implements the architecture.

The most basic level of dynamic change in the address mapping tables is a change to a segment descriptor. This occurs when a segment is moved (to another primary memory, or onto secondary storage) or when it is locked during a multi-memory reference operation. This problem was examined, in a broader context than Cm*, in Chapter 5. The techniques proposed in Chapter 5 have been shown to work effectively and in a natural manner on Cm*. One technique is to have a single point of object definition (S → M mapping). This corresponds to the single logical copy of a segment descriptor in Cm*. The other technique is to associate "Location Anticipation" information with the Ex-Env to System name space mapping. In Cm*, the expected location of a segment is associated with a Window Register when a segment is made addressable.

An asynchronous, dynamic change is also possible at another level in the address mapping structures. As currently defined, a capability from a shared C-list can be deleted at any time. We have seen that correctly implementing this is straightforward with a single cluster, but is substantially more complex for a full multi-cluster implementation. However, an efficient implementation with multiple clusters where the final capability in an addressing chain can be asynchronously deleted is quite feasible and probably will actually be implemented.

It is disturbing that the present architecture and operating system do not explicitly

prevent the deletion of a capability which is a non-final component in an addressing chain. For example, if a process deletes a capability from its own Primary C-list which is part of some addressing path, then the path will be correctly invalidated. However, if some other process--or the kernel executing on another processor-- deletes the same capability, the addressing path will not be invalidated until a context swap occurs. A correct implementation, which would allow this sort of dynamic change in a multi-cluster system, would be very slow and cumbersome. Allowing asynchronous changes to a descriptor is relatively easy, because a single logical copy can exist close to the physical memory referred to by the descriptor. All paths to that segment must go to that memory, and hence can all go through the descriptor with little or no extra cost. Supporting asynchronous changes to one level of capabilities is feasible because a single cached copy can be placed adjacent to the descriptor referred to by the capability. However, changes to the capability must be reflected both in the cluster containing the C-list and the cluster containing the cached copy.

For capabilities occurring at higher levels in the mapping sequences, there is no convenient single location to hold a cached copy. Mapping sequences through that capability may originate in any cluster. These sequences can terminate in the cluster of any segment reachable via that capability. If a single cached copy exists, then references using that capability must all indirect via the cluster containing the copy. It is possible, that a reference by a processor to a segment in its own local memory, would have to be made via one or more other clusters. If multiple cached copies are permitted, then updating the capabilities becomes extremely difficult.

These problems occur when a process has delete rights on a C-list whose entries may form part of an address mapping chain for some other process. As suggested in Section 6.9.1, there is no necessity to permit this situation. If it is necessary that a capability be both shared and deletable, then it can be copied to a private C-list before being used as part of an addressing chain. Various methods, using the present facilities for segment typing and rights restrictions on capabilities, can be proposed to enforce this restriction.

Alternatively, a very flexible scheme results from associating a reference count with a capability. Capabilities represented in C-lists, where they might be part of an addressing chain, would be expanded from two to three words to hold a 16 bit reference count. The reference count would be incremented when the capability was de-referenced (an implicit copy) as part of a mapping sequence and decremented when that addressing path was broken. A capability with a non-zero reference count could not be deleted. Note that capabilities stored in mail boxes, etc. where they cannot be used for addressing need occupy only two words--since by definition their reference count must be zero.

This change would be transparent to user programs, and most of the kernel, on STAROS. It would substantially simplify the microcode necessary to implement the architecture on multiple clusters. The present potential for unpredictability when a capability is deleted would be corrected.

6.12 Conclusions

We have seen that the hardware structure of Cm* is capable of supporting a powerful addressing architecture which allows close, protected cooperation between many, parallel executing processes. Close cooperation during concurrent execution is supported through shared access to segments and shared capability lists. Other forms of cooperation are supported by a fast, interprocess, message facility. The message passed is a capability, so both individual data segments and entire data structures can be passed in a protected way without copying. Support is also given to assist state saving and restoring--the effectiveness of this is limited by characteristics of the LSI-11. We have examined the problems in implementing this architecture on a parallel, distributed structure. A single cluster version is operational and supports the operating system STAROS. We have described the mechanisms necessary for extending the implementation to multiple clusters.

7. Summary and Conclusions

7.1 Introduction

Cm* is an extensible, multiprocessor computer system with a hierarchical structure. A 10 processor pilot system, constructed in the Computer Science Department, has been in operation for a year. A 50 processor system will be operational late in 1978. The hardware structure will support on the order of 10,000 processors. A major overall design objective has been to efficiently support close cooperation between large numbers of concurrently executing processes.

An important component of multiprocessor systems is the switching structure which allows processors to access shared memory. The effective, maximum size (in number of processors) of multiprocessors described in the literature is limited. This limitation comes either through saturation of the access path to shared memory, or through the rapid growth of switch cost. The hierarchical switching structure of Cm* offers, in principle, indefinite extensibility of processing power, memory capacity and communication bandwidth. The cost of the interconnection structure grows approximately linearly with system size. Effective use of the structure depends on suitable decompositions of applications. We give the motivations for the switching structure and describe numerous techniques used to allow a high performance, cost-effective, deadlock free, switch realization.

The addressing architecture of a computer system has a strong influence on its programmability. The addressing architecture is important both at the level of naming operands, for example distinguishing between stack and register oriented designs, and in naming larger conceptual entities, such as segments. The former level is determined primarily by the instruction set design. The latter level is primarily determined by the operating system. The work in this thesis is concerned with understanding addressing architecture at the level of naming and sharing objects. An object is the smallest logical collection of information (for example a segment) which is independently manipulated by the operating system and can be directly accessed by a program.

We present a simple model which allows the systematic analysis and comparison of addressing architectures. The creation of an independent addressing environment when each subprogram is invoked (as in HYDRA [Wulf et al, 74] and FAMOS [Habermann et al, 76]) is compared to process-based addressing (as used in Multics [Organic, 72] and CAP [Needham, 72]). We show that independent addressing environments for each subprogram allows greater flexibility in the time of name binding, has fewer levels of indirection, and is better suited to multiprocessor systems.

Close cooperation and communication between processes requires the sharing of objects--shared access to data, semaphores, mailboxes, etc. Proper support of this sharing (particularly in a system with many, concurrently executing processes) is shown to have a substantial impact on the switching structure and addressing hardware of a multiprocessor.

As viewed from an individual processor, Cm* has a non-homogeneous structure. The access path and access time from a processor to a particular memory depends on their relative position in the structure. This inhomogenity could reduce the programmability of the system. However, the Cm* addressing architecture presents to the programmer a uniform, system wide, segmented address space. Segments are typed, protected objects. All references, by both user and kernel programs, are via capabilities. Capabilities may be passed between processes via mailbox segments. This allows the protected passing of messages, including large data structures represented by a capability list, without copying. Numerous facilities are provided in the architecture to support operating system primitives. Special provision is made to allow the movement of a segment without requiring every processor to update its mapping tables.

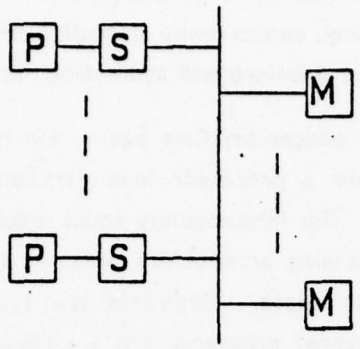
7.2 The Switching Structure of Multiprocessors

As used in this work, the term *multiprocessor* means a collection of independent processors which communicate through shared access to primary memory. An important aspect of their design is the switching structure which gives processors access to shared memory. Most multiprocessors have some form of crosspoint switch. For example, the fully distributed switch of Pluribus [Heart et al, 73], the multiported memory of dual PDP-10's, etc. and the fully centralized switch of C.mmp [Wulf and Bell, 72]. The major drawback of crosspoint switches is that their complexity, and hence cost, grows at N^2, where N is the number of processors and memories. Multiprocessor structures have been proposed which share access to memory via a single bus. The complexity of this grows at N, but the overall performance is limited by the bandwidth of a single bus.

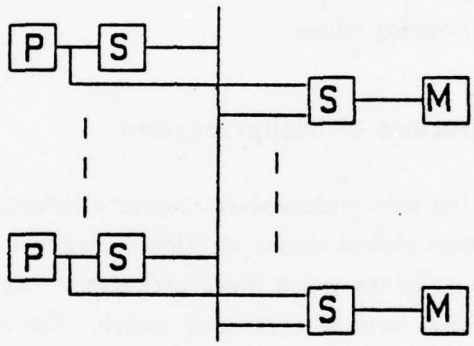
One method of deriving the Cm* switching structure is shown in Figures 7-1(a - d) and 7-2(e - f). Switch complexity grows at approximately 2N. If all processors access only their local memory, then the total memory bandwidth is the same as for a full crosspoint switch[1]. If all processors access memory local to some other processor, then performance is limited by the bandwidth of the interprocessor buses. Thus effective use of the system depends upon exploiting program locality.

---

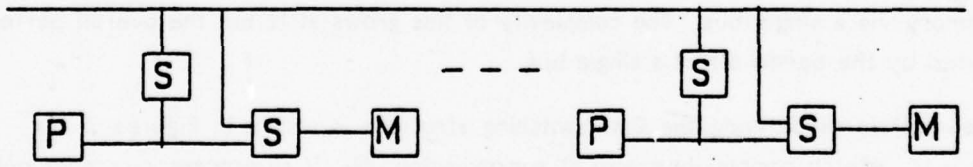
[1] Possibly higher, because the simple switches in the Cm* structure can have a shorter delay than a large crosspoint switch.



(a) All Processors have Access to All Memory



(b) Add Private Data Path to Designated Memory



(c) Rearrange the Representation of the Structure

Figure 7-1: Derivation of the 'Cm* Switching Structure (Part 1)

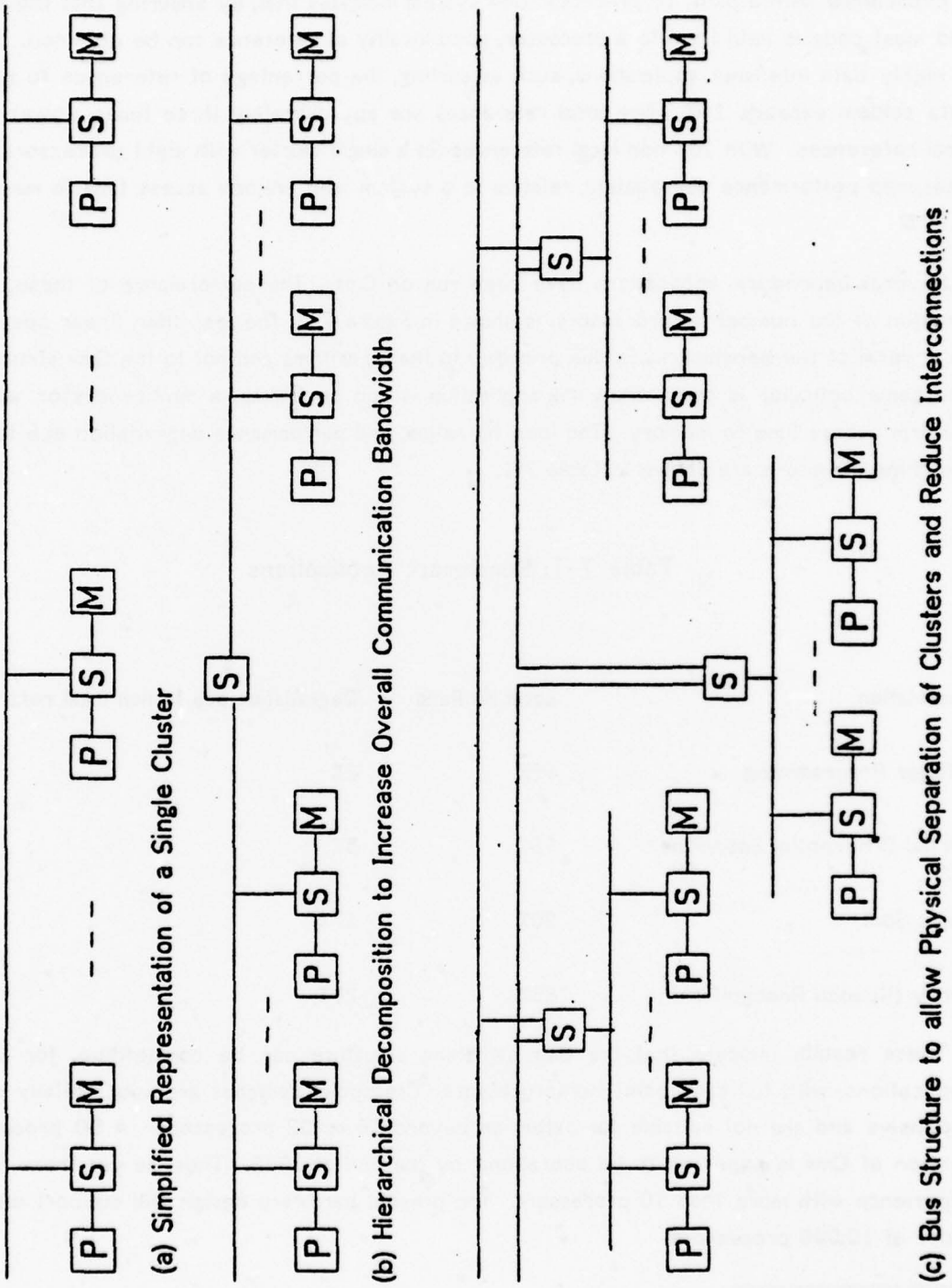


Figure 7-2: Derivation of the Cm* Switching Structure (Part 2)

Experience with a pilot, 10 processor Cm* system indicates that, by ensuring that the stack and most code is held local to a processor, good locality of reference can be obtained. Even in highly data intensive applications, such as sorting, the percentage of references to global data seldom exceeds 10%. Non local references are approximately three times slower than local references. With 10% non local references (in a single cluster with eight processors) the measured performance degradation, relative to a system with uniform access time to memory, is 17%.

Several benchmark applications have been run on Cm*. The performance of these, as a function of the number of processors, is shown in Figure 7-3. The less than linear speedup, on several of the benchmarks, is due primarily to the algorithms and not to the Cm* structure. The same behavior is seen when the application is run on C.mmp, a multiprocessor with a uniform access time to memory. The local hit ratios, and performance degradation due to the switching structure are shown in Table 7-1.

Table 7-1: Benchmark Applications

Application	Local Hit Ratio	Degradation due to non local refs.
Integer Programming	99%	2%
Partial Differential Equations	98%	5%
Quick Sort	90%	17%
Harpy (Speech Recognition)[1]	85%	23%

These results indicate that the Cm* switching structure can be competitive, for some applications, with full crosspoint multiprocessors. Crosspoint switches are substantially more expensive and are not suitable for extension beyond 16 or 32 processors. A 50 processor version of Cm* is expected to be operational by the end of 1978. Thus, as yet there is no experience with more than 10 processors. The present hardware design will support on the order of 10,000 processors.

[1] The local hit ratio could be improved to be about 91% by duplicating some global, read only data.

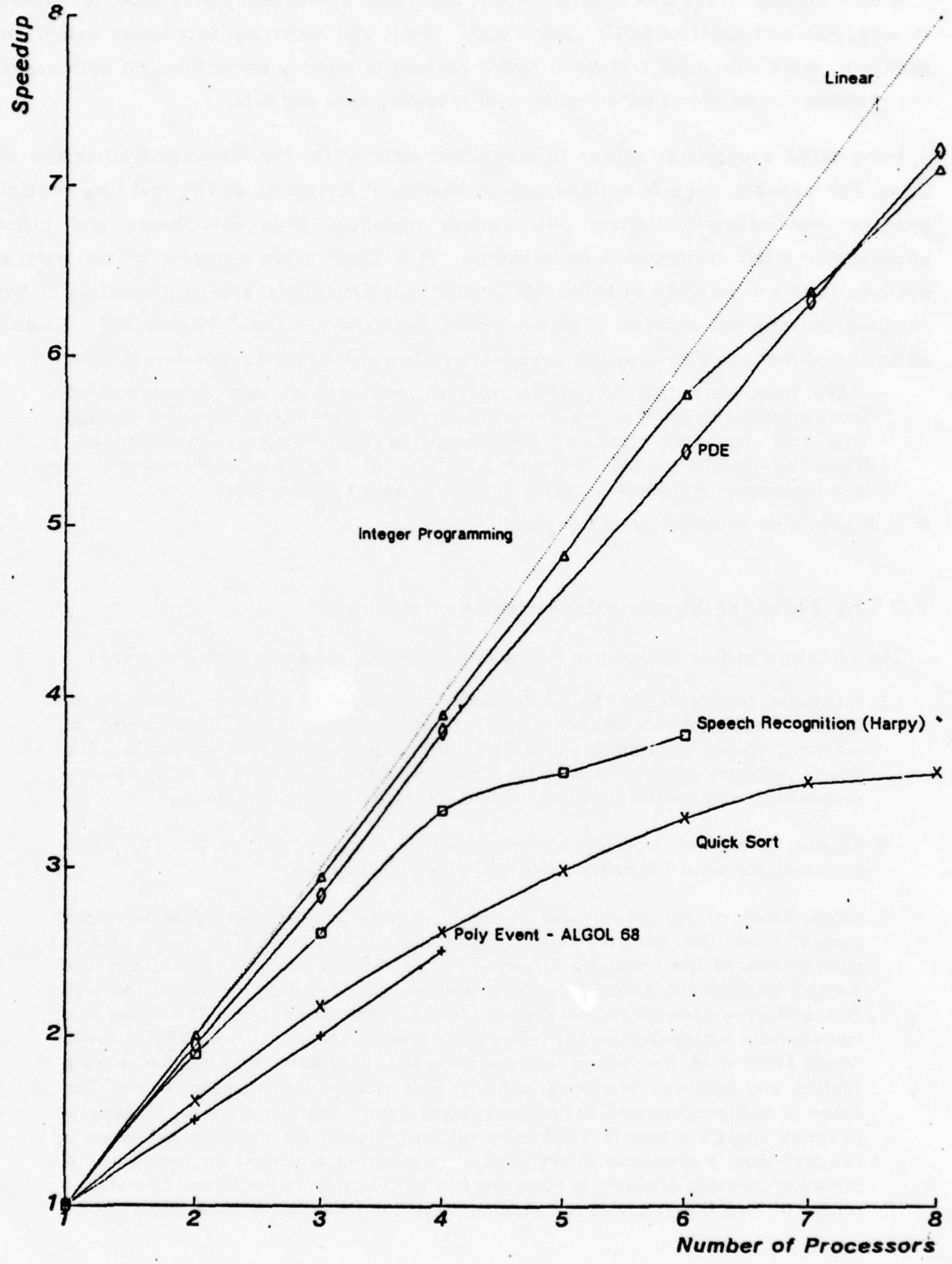


Figure 7-3: The Average Speed-up of Four Algorithms on Cm*

A disadvantage of the Cm* structure is that duplication of frequently used code, to achieve locality, may add substantially to system cost. This is also necessary, to a lesser extent, on multiprocessors with a full crosspoint switch because of memory contention. In both cases, the problem can be alleviated with caching of read-only code and data.

Many large applications appear to decompose suitably for the hierarchical structure of Cm*: For example, large AI systems such as Hearsay II [Lesser et al, 75], real time control, program verification [Jefferson, 78], weather modelling, large data bases and other applications often implemented on networks. H. A. Simon gives examples of hierarchical systems from a wide range of fields. He asserts that, "Empirically, a large proportion of the complex systems we observe in nature exhibit hierarchic structure" [Simon, 62]. Simon's definition of hierarchic systems has a remarkably close match to the properties of Cm*.

> We have seen that hierarchies have the property of near decomposability. Intra-component linkages are generally stronger than inter-component linkages. This fact has the effect of separating the high-frequency dynamics of a hierarchy--involving the internal structure of the components--from the low-frequency dynamics--involving interaction among components.

H. S. Simon--The Architecture of Complexity, 1962.

7.3 The Switching Structure Implementation of Cm*

The structure of Cm* is shown in Figure 7-4. The major components of Cm* are:

1. **Computer Module or Cm.** This is the basic building block of Cm*. It consists of Digital Equipment Corporation LSI-11 microprocessor, up to 256K bytes of memory, device interfaces, and a CMU built local switch, called an Slocal. The Slocal provides the interface to the Map Bus, allows relocation of local memory references, and control functions such as interprocessor interrupts.

2. **Cluster.** Computer modules are combined in clusters. of up to 14, which communicate via a special purpose bus, called the **Map Bus.**

3. **Kmap.** Much of the power and flexibility of the Cm* addressing architecture comes from the central mapping controller in each cluster. The major components of the Kmap are shown in Figure 7-5. All non local references are passed through the Kmap. The **Kbus** performs all arbitration for use of the Map Bus and generates all control signals for the synchronous bus. The **Pmap** is a horizontally microprogrammed, 150ns cycle time processor. It has many features which tailor it to the task of address mapping: field extraction logic, elaborate testing and multiway branching, and fast data storage for mapping tables. The Pmap is multiprogrammed, in hardware, eight ways. There is a set of 32 general purpose registers and 16 subroutine return registers for each **context.** When a request from a processor is accepted, it is allocated a context by the Kbus. A queue of runable contexts is maintained to the Pmap from the Kbus. Likewise, a queue of Map Bus requests is maintained from the Pmap to the Kbus. This allows

concurrency in the mapping mechanism. The context mechanism also facilitates error recovery and reporting. The Pmap has 4K words (by 80 bits) of writable control store. The implementation of a segmented addressing architecture, for a single cluster, requires 1500 words. A simpler architecture, which supports multicluster systems, requires 500 words. In the simpler architecture six microinstructions are executed for a typical reference. In the more powerful architecture eight microinstructions are executed [Ousterhout, 77].

4. Linc. The Linc is a component of the Kmap which provides communication with other clusters via two intercluster buses. A typical, three word message can be transferred between clusters in less than 2us.

A within cluster reference is shown in Figure 7-6, an inter-cluster reference is shown in Figure 7-7. The interference time, for an LSI-11 to local memory, is approximately 3.3us. This is unaffected by the CMU modifications. Interference time for intra-cluster references is approximately three times slower, 9.3us. Interference time for references to an adjacent cluster is 24us.

7.3.1 Deadlock in the Switching Structure

Unlike more conventional multiprocessors, the Cm* structure is susceptible to deadlock. Chapter 3 of the thesis explores the source of this deadlock potential and describes the techniques adopted to prevent deadlock. Resources, over which deadlock can occur, include buses, contexts in the Pmap and buffers used for inter-cluster communication. The deadlock prevention algorithms used in operating systems [Coffman and Denning, 73] are not directly applicable because Cm* is a distributed structure without central control. In addition, these resources are reallocated at memory reference rates and no computational overhead for deadlock avoidance can be tolerated.

7.3.2 Packet Switching and Processor Modifications

Above the local LSI-11 memory bus level, Cm* is implemented as a packet switched network. In conventional systems, a full path--or circuit--is established between a processor and memory unit during a memory reference. In Cm*, this would allow deadlock over bus allocation and leads to inefficient use of buses. The techniques used in Cm* to implement packet switching, at the memory reference level, are described in Chapter 4. The structure of a deadlock free, single cluster Cm*-like system, implemented with circuit switching, is sketched in Chapter 3. The Map Bus and Inter-cluster Bus have comparable basic data rates, about 10 MHz, but use substantially different protocols. We discuss the design criteria for these buses.

Conventional processors, such as the LSI-11, require substantial modifications to be incorporated into Cm* without allowing deadlock. These modifications, and other additions to

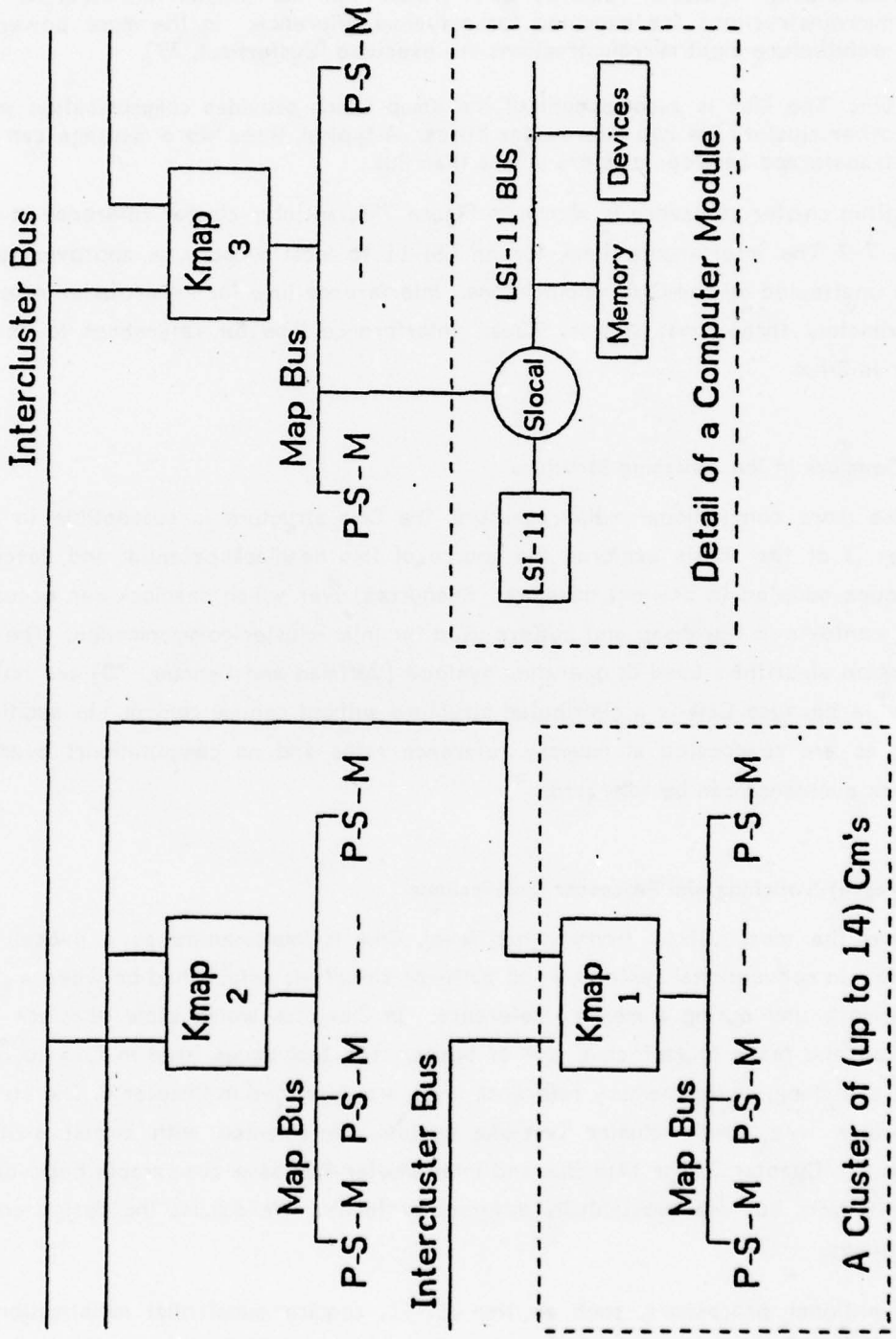


Figure 7-4: A Simple, 3 Cluster Cm* System

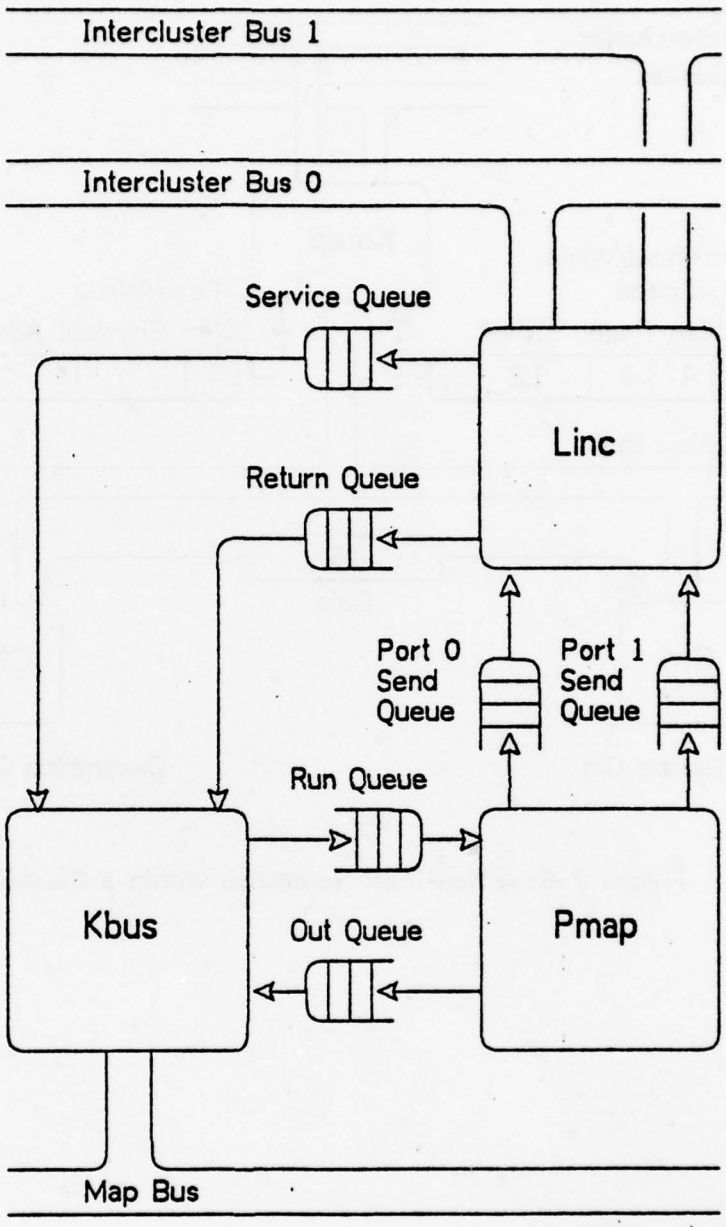


Figure 7-5: The Components of the Kmap

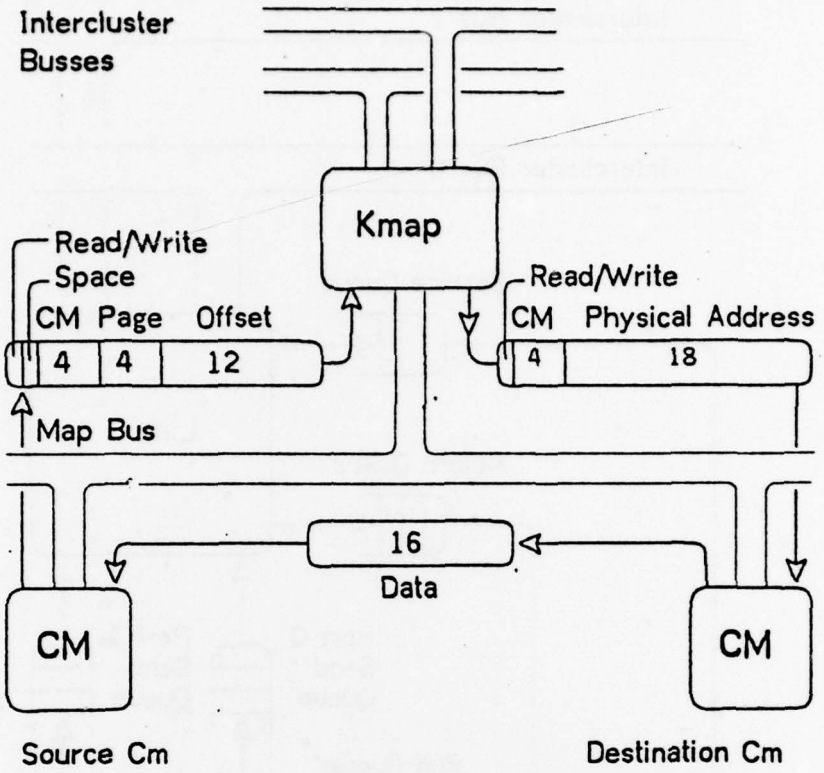


Figure 7-6: A Non-local Reference within a Cluster

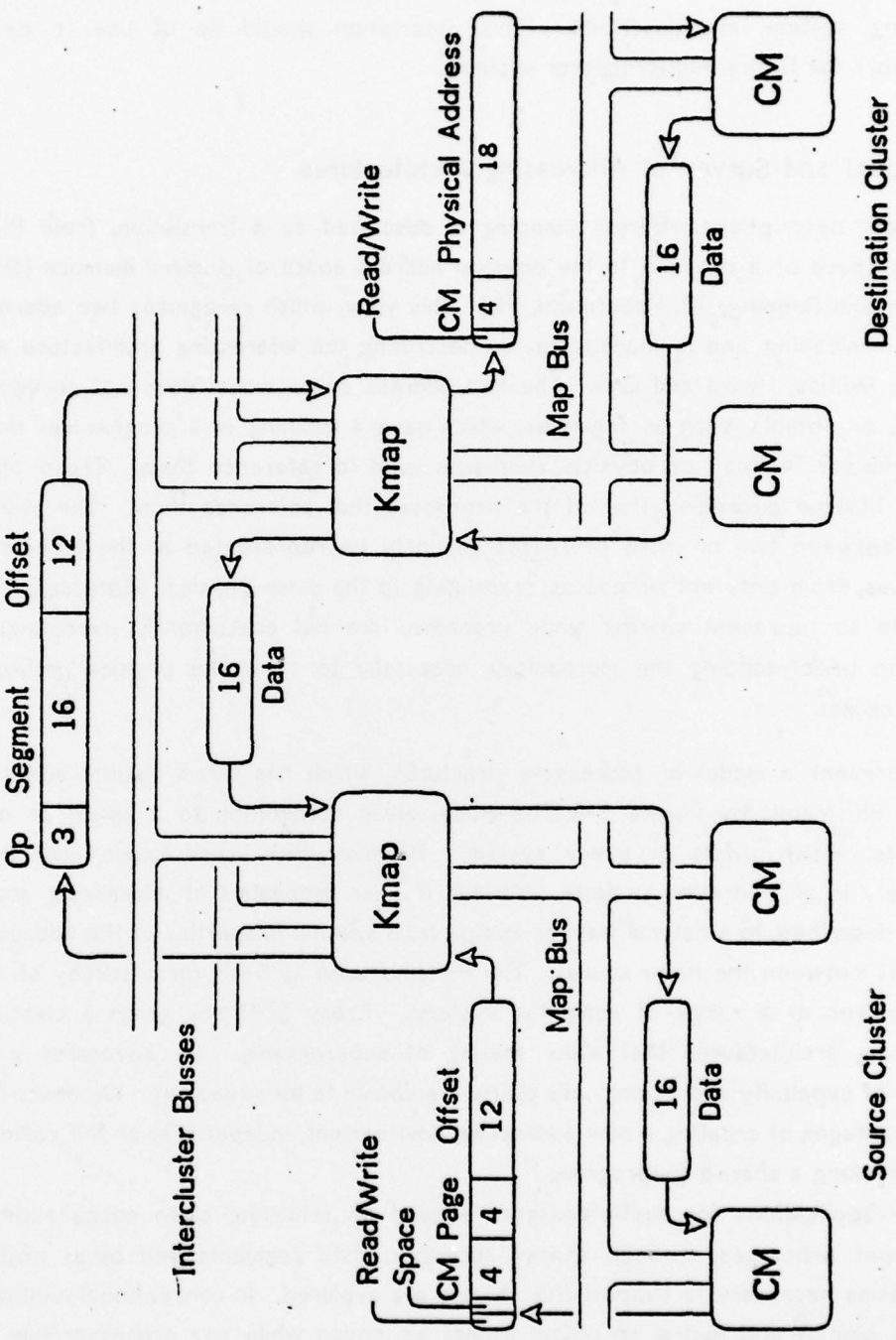


Figure 7-7: An Intercluster Reference

the LSI-11's to support interprocessor control primitives and give basic protection to the operating system, are described. This description should be of use to designers of processors for future multiprocessor systems.

7.4 Model and Survey of Addressing Architectures

In most descriptions, address mapping is described as a translation, from the "virtual" address space of a process, to the physical address space of primary memory [Denning, 70; Coffman and Denning, 73; Habermann, 76]. This view, which recognizes two address spaces, can be misleading and is inadequate for describing the addressing architecture of systems such as Multics, Hydra and Cm*. The two address space model does not recognize logical entities, or objects, such as segments, which have a meaning to a programmer unrelated to the particular "virtual" or physical addresses used to reference them. These objects may have a lifetime exceeding that of the processes that reference them. The sharing of an object between two or more processes can only be represented at the level of "virtual" addresses, from different processes, translating to the same physical addresses. This is not adequate to represent sharing when processes are not concurrently executing; nor is it useful in understanding the mechanisms necessary to allow the physical movement of a shared object.

We present a model of addressing structures, which has three distinct address spaces. This is illustrated by Figure 7-8. The model gives recognition to a space of objects, or segments, which exists in every system. However, this name space may not appear explicitely in all operating systems. Various familiar properties of addressing architectures can be described, in a natural way, as arising from specific properties of the address mapping functions between the name spaces. The model is used as basis for a survey of addressing architectures of a range of computer systems. Fabry [74] has given a classification of addressing architectures that allow sharing of subprograms. He advocates a particular version of capability addressing. His claims are shown to be misleading. We describe some of the advantages of creating a new addressing environment, independent of the calling process, when invoking a shared subprogram.

Many applications for multiprocessors depend on achieving close cooperation, between concurrent processes, through shared access to data segments and other objects. The mechanisms necessary to support this sharing are explored. In conventional multiprocessors, such as Multics and Hydra, an object cannot be moved while any processor has it directly addressable. If this restriction were carried over to systems with 50, or 50,000 processors it would make demand paging, dynamic reconfiguration, etc. infeasible. Specific techniques to avoid this problem are presented and the consequences for the physical structure of multiprocessors explored.

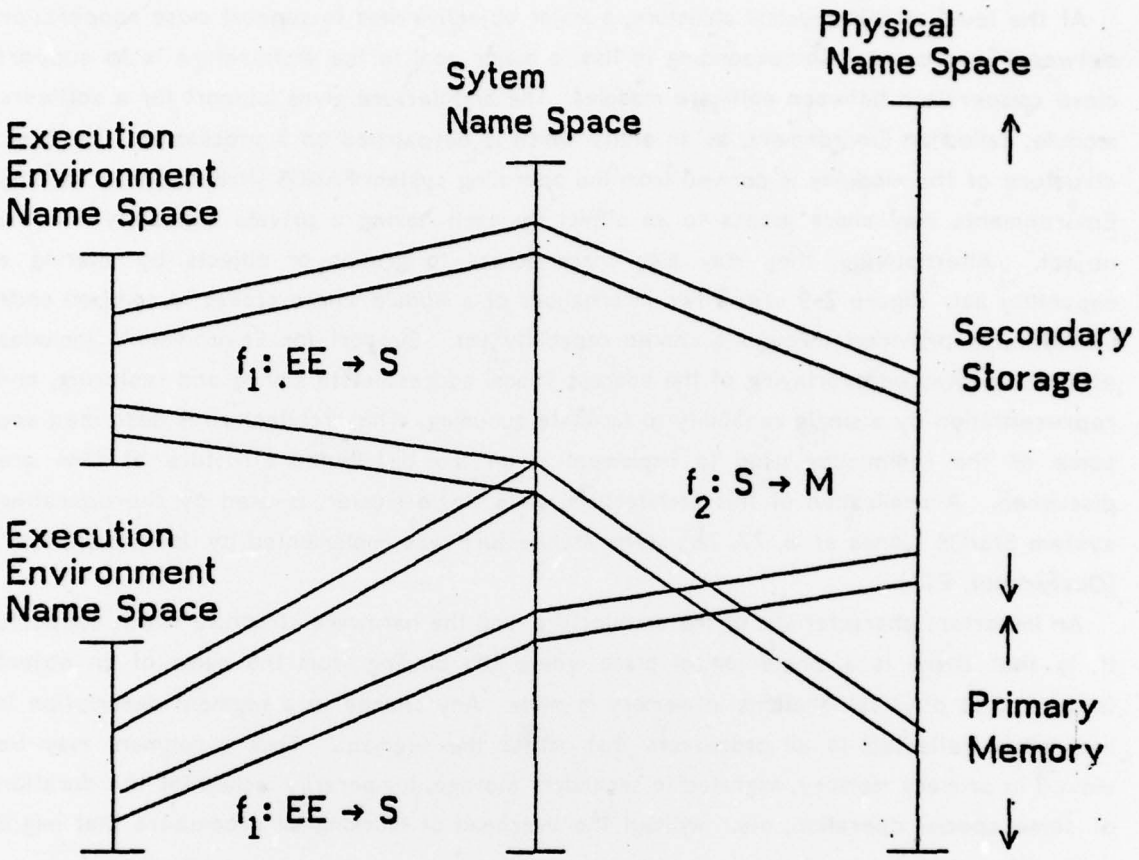


Figure 7-8: The Three Name Space Model of Addressing Structures

7.5 The Addressing Architecture of Cm*

At the hardware level, Cm* is a distributed, non-homogeneous structure with 16 bit microcomputers. A systems programmer, however, is presented with a uniform, 28 bit, segmented virtual address space. Segments can only be addressed via capabilities, which provide a fine grain control over memory sharing and the protected passing of messages without copying. Segments are a simple form of typed object. Segment types include: data, capability list, descriptor directory, control stack, mailbox and user environment. The Cm* hardware structure allows invocation of protected operations on these objects with low overhead. (For example, a capability may be transferred, in a fully protected way, between

capability lists in the execution time of approximately 10 LSI-11 instructions.)

At the level of the physical structure, a major objective was to support close cooperation between processors. Corresponding to this, a major goal in the architecture is to support close cooperation between software modules. The architecture gives support for a software module, called an Environment, as an entity which is despatched on a processor. The basic structure of the modules is derived from the operating system FAMOS [Habermann et al, 76]. Environments may share access to an object by each having a private capability for the object. Alternatively, they may also share access to groups of objects by sharing a capability list. Figure 7-9 shows two incarnations of a module where access to common code and data is provided through a shared capability list. Support for Environments includes efficient, protected overlaying of the address space, address state saving and restoring, and representation by a single capability to facilitate queueing. This architecture is described and some of the techniques used to implement it on the distributed structure of Cm* are discussed. A realization of this architecture, on a single cluster, is used by the operating system StarOS [Jones et al, 77, 78]. (The architecture was implemented by John Ousterhout [Ousterhout, 77].)

An important characteristic of the architecture, and the hardware structure which supports it, is that there is a single logical place where the binding from the name of an object (segment) to physical locations in memory is made. Any change to a segment description is indivisibly reflected to all processors that access the segment. Thus a segment may be moved in primary memory, migrated to secondary storage, temporarily locked for the duration of some special operation, etc. without the overhead of blocking all processors that might access it.

7.6 Conclusion

The Hierarchical structure of Cm* has shown a unique way to interconnect processing elements and memories. It combines many of the advantages of computer networks (extensibility, low interconnection costs, etc.) with many of the programming conveniences of a more conventional multiprocessor (close interactions, resource sharing, etc.). Cm*/50, with 50 processors, will be substantially larger than comparable systems described in the literature. Even without hardware changes, the structure could support on the order of 10, 000 processors.

Although we have presented some detailed performance measurements and reviewed the performance of several benchmark programs, it is very difficult to objectively evaluate a non-standard computer architecture. The use of a multiprocessor, requires programmers to think in a new way. We have had very little experience in applying multiprocessors, of any

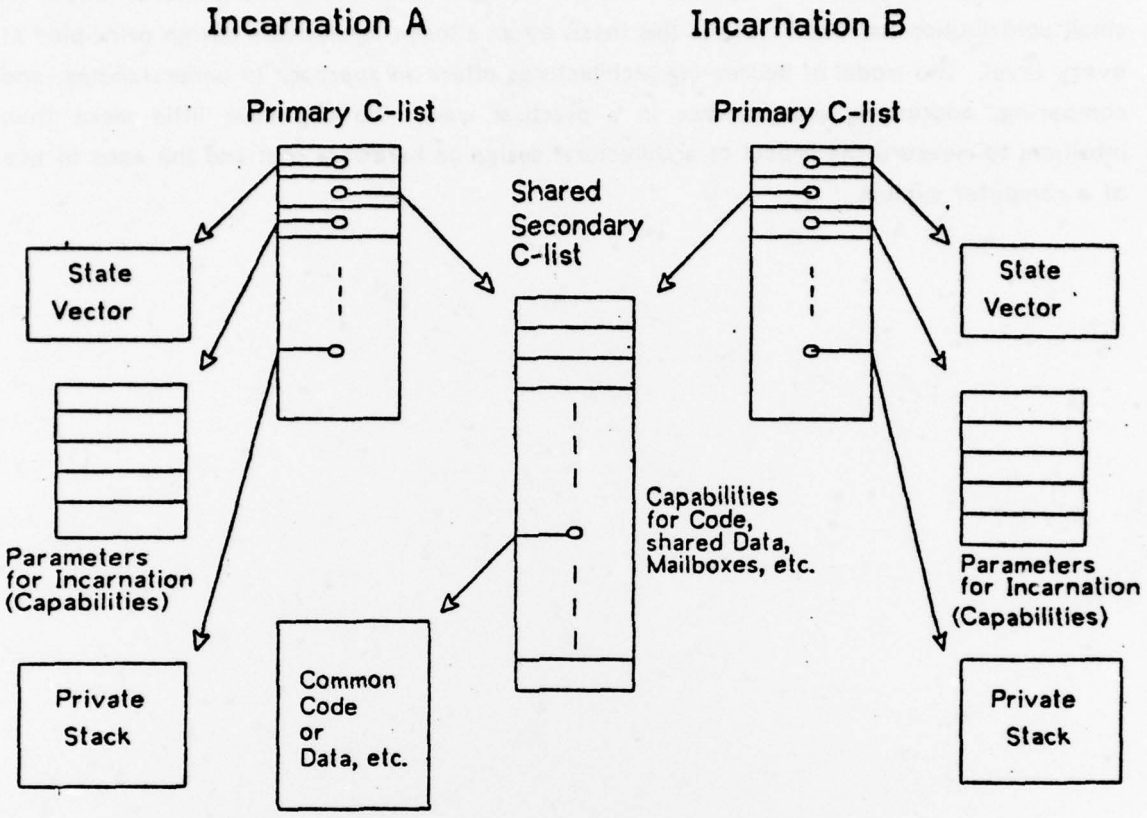


Figure 7-9: Two Module Incarnations

sort, to practical problems. We have the technology to, economically, construct multiprocessors with almost any desired number of processing elements. However, much research work remains before it will be commonplace to apply 10,000 processors to a single task. Many of the research issues lie at the levels of operating systems, languages and algorithms.

In the near future, structures similar to Cm* will have commercial application in dedicated systems with a small number of processors. The commercial attraction of the systems will be extensibility and reliability, rather than total performance or cost-performance.

As an area of human endeavor, a great deal of creative energy has gone into the design of computer systems. However, very little attempt has been made to systematically study and

understand the principles of computer design at the structural and architectural level. A small contribution has been made in this thesis by an attempt to elucidate design principles at every level. The model of addressing architectures offers an approach to understanding, and comparing, addressing architectures in a practical way. We still have little more than intuition, to measure the impact of architectural design on hardware cost and the ease of use of a computer system.

Bibliography

[Balas and Padberg, 76] Balas, E. and M. Padberg, "Set partitioning-A survey," *SIAM Review*, vol.18, no. 4, Oct. 1976.

[Barnes et al,68] Barnes, G., R. Brown, M. Kalo, D. Kuck, D. Slotnick, and R. Stokes, "The ILLIAC IV computer," IEEE Trans. Comput., vol. c-17, Aug. 1968.

[Baudet, 76] Baudet, G., "Asynchronous Iterative Methods for Multiprocessors," Tech. Report, Dept. Computer Science, Carnegie-Mellon University, Pittsburgh, Pa. 15213, Nov. 1976.

[Bell and Newell, 71] Bell, C. G. and A. Newell, "Computer Structures: Reading and Examples," New York: McGraw Hill, 1971

[Benes, 65] Benes, V. E., "Mathematical Theory of Connecting Networks and Telephone Traffic," *Academic Press*, New York. 1965

[Chansler, 76] Chansler, Jr., R. J., "Cm* simulator users manual," Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa. 15213 1976.

[Chen, 74] Chen, Robert., "Bus Communication Systems," PhD Thesis, Dept. Computer Science, Carnegie-Mellon University, Pittsburgh, Pa. 15213. January 19 1974.

[Coffman and Denning, 73] Coffman, Edward, and Peter Denning, "Operating Systems Theory," Prentice-Hall, 1973.

[Courtois, 1977] Coutois, P. J., "Decomposability, Queueing and Computer Systems Applications," *Academic Press*, New York, 1977

[Daley and Dennis, 73] Daley, R. C., and Jack Dennis, "Virtual Memory Processes, and Sharing in MULTICS", Comm. ACM, May 68.

[Denning, 70] Denning, P. J., "Virtual Memory," *Computing Surveys*, 2, 3, (Sept. 1970).

[Fabry, 74] Fabry, R. S., "Capability Based Addressing," *CACM 17, 4* (July 1974)

[Flon, 75] Flon, Larry, "Program Design with Abstract Data Types", Tech. Report, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa. 15213. 1975

[Fuller, 76] Fuller, S. H., "Price/performance comparison of C.mmp and the PDP-10," IEEE/ACM Symp. on Comput. Architecture, pp. 195-202, Jan. 1976.

[Fuller et al, 77] Fuller, S.H., A. K. Jones, and L. Durham, Eds., "Cm* review, June 1977," Tech. Report, Dept. Computer Science, Carnegie-Mellon University, Pittsburgh, Pa. 15213, June 1977.

[Fuller and Oleinick, 76] Fuller, S. H. and P. N. Oleinick, "Initial measurements of

parallel programs on a multi-miniprocessor," 13th IEEE Comput. Soc. Int. Conf., Washington, DC, Sept. 1976.

[Fuller et al, 78] Fuller, S. H., J. K. Ousterhout, L. Raskin, P. I. Rubinfeld, P. J. Sindhu, R. J. Swan, "Multimicroprocessors: An Overview and Working Example," *Proc. IEEE* Jan. 1978.

[Gavish et al, 77] Gavish, B., P.M. Merlin, P.J. Schweitzer, "Minimal Buffer Requirements for Avoiding Store-and-Forward Deadlock," Research Report, IBM Yorktown Heights, Aug.1977.

[Gunther, 75] Gunther D., "Prevention of Buffer Deadlocks in Packet-switching Networks," report presented at IFIP-IIASA Workshop on Data Communications, Laxenburg, Austria, Sept. 15-19, 1975.

[Habermann, 76] Habermann, A. Nico, "Operating Systems," SRA, 1976.

[Habermann, 69] Habermann, A. Nico, "Prevention of System Deadlocks", Communications of the ACM, July 1969.

[Habermann et al, 76] Habermann, A. N., L. Flon, L. Cooprider, "Modularization and Hierarchy in a Family of Operating Systems," *CACM*. May 1976.

[Heart et al, 70] Heart, F. E., Kahn, R. E., Ornstein, S. M., Crowther, W. R., and Walden, D. C. "The Interface Message Processor for the AR ARPA Computer Network," *American Federation of Information Processing Societies Proceedings, SJCC*, vol. 36 (1970), pp 551-567.

[Heart et al, 73] Heart, F. E., Ornstein, S. M., Crowther, W. R., and Barker, W. B. "A new Minicomputer/Multiprocessor for the ARPA Computer Network," *American Federation of Information Processing Societies, Proc. NCC*, vol. 42 (1973)

[Herbert, 78] Herbert, A. J., "A Hardware Supported Protection Architecture", University of Cambridge, England. To be published.

[Hibbard et al, 78] Hibbard, P., A. Hisgen, T. Rodeheffer, "A Language Implementation Design for a Multiprocessor Computer System, " *IEEE 5th Annual Symposium on Computer Architecture*, pp 66-72 1978

[Holt, 72] Holt, Richard, "Some Deadlock Properties of Computer Systems", *Computing Surveys*, Sep 72.

[Jefferson, 78] Jefferson, D. Private Communication.

[Jensen and Anderson, 77] Jensen E. D., and G. A. Anderson, "Computer interconnection structures: Taxonomy, characteristics, and examples," *Computing Surveys*, vol 7, no. 4, Dec. 1977.

[Jones et al, 77] Jones, A. K., R. J. Chansler, I. Durham, P. Feiler, K. Schwans. "Software Management of Cm*, a distributed multiprocessor,"

AFIPS Conf. Proc., vol. 46, 1977 National Computer Conference.

[Jones et al, 78] Jones, A. K., R. J. Chansler, I. Durham. P. Feiler, D. Scelza, K. Schwans, S. Vegdahl. "Programming Issues raised by a Multiprocessor," *Proc. IEEE*, vol. 66, 2 February 1978.

[Lesser et al, 75] Lesser, V. R., R. D. Fennell, L.D. Erman, D.R. Reddy, "Organization of the Hearsay II Speech Understanding System, " IEEE Trans. ASP-23, No 1. Feb. 1975.

[Liskov, and Zilles, 74] Liskov, Barbara, and Zilles, Stephan "Programming with abstract Data Types", *SIGPlan*, April 74.

[Lowerre, 76] Lowerre, B. T., " The Harpy Speech Recognition System," Carnegie-Mellon University, Computer Science Dept. Technical Report Apr. 1976.

[McWilliams et al, 77] McWilliams, Thomas M., Lawrence C. Widdoes, Jr., and Lowell L. Wood, "Advanced Digital Processor Technology Base Development for Navy Applications: The S-1 Project", Sept. 30, 1977, UCID-17705, Lawrence Livermore Laboratory.

[Needham, 72] Needham, R. M. "Protection Systems and protection Implementation," *Proc. AFIPS 1972 FJCC* Vol. 41.

[Newell and Robertson, 75] Newell, A., and G. Robertson, "Some issues in programming multi-miniprocessors," *Behavior Research Methods and Instrumentation*, Vol. 7, no. 2, March 1975.

[Organick, 72] Organick, E. I. "The Multics System: An Examination of its Structure," MIT Press, 1972.

[Ousterhout, 77] Ousterhout, J. "Kmap Microprogram," in *Cm*: Review*, June 1977 CMU Tech Report.

[Parnas, 72] Parnas, D. L. "On the criteria to be used in decomposing systems into modules," Communications of the Association for Computing Machinery, vol 15, Dec. 1972.

[Scelza, 77] Scelza, D., The Cm* Host User Manual. Dept. of Comput. Sci., Carnegie-Mellon University, Pittsburgh, PA. July 1977.

[Siewiorek, 77] Siewiorek, D. P., "Multiprocessors: Reliability Modelling and Graceful Degradation," *Infotech State of the Art Conference: System Reliability*, London, UK. May 1977.

[Siewiorek et al, 78a] Siewiorek, Daniel P., Vittal Kini, Henry Mashburn, Stephen McConnel, Michael Tsao, "A Case Study of C.mmp, Cm*, and C.vmp -- I. Experiences with Fault Tolerance in Multiprocessor Systems", to appear in *Proceedings of the IEEE*, Volume 66 Number 10, October 1978.

[Siewiorek et al, 78b] Siewiorek, Daniel P., Vittal Kini, Rostam Joobbani, Harold Bellis, "A Case Study of C.mmp, Cm*, and C.vmp -- II. Predicting and

Calibrating Reliability of Multiprocessor Systems", to appear in *Proceedings of the IEEE*, Volume 66 Number 10, October 1978.

[Simon, 62] Simon, H. A. "The Architecture of Complexity," reproduced in *Sciences of the Artificial* MIT Press. 1974.

[Sutherland and Mead, 77] Sutherland, I. and C. Mead. "Microelectronics and Computer Science," *Scientific American*, September 1977

[Swan et al, 76] Swan, R. S., S. H. Fuller, D. P. Siewiorek. "The Structure and Architecture of Cm*: A Modular, Multi-Microprocessor," *The Computer Science Department Research Review 1975 - 1976*. CMU, December 1976.

[Swan et al, 77a] Swan, R. J., S. H. Fuller, D. P. Siewiorek. "Cm*: a Modular, Multi-Microprocessor," *AFIPS Conf. Proc.* Vol. 46, 1977 National Computer Conference.

[Swan et al, 77b] Swan, R. J.,A. Bechtolsheim, K. Lai,. J. Ousterhout. "The Implementation of the Cm* Multi-Microprocessor," *AFIPS Conf. Proc.* Vol. 46, 1977 National Computer Conference.

[Tandem, 77] "Tandem 16 System Introduction," Tandem Computers, Cupertino, CA 95014 1977.

[Thompson, 77] Thompson, C. D., "Generalized Connection Networks for Parallel Processor Intercommunication," Tech Report, Dept. Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., May 1977.

[Wulf et al, 74] Wulf, W. A., et al. "HYDRA: The Kernel of a Multiprocessor Operating System," *Comm. of the ACM*, Vol. 17, June 1974.

[Wulf and Bell, 72] Wulf, W. A., C. G. Bell. "C.mmp-a multi mini processor," *AFIPS Conf. Proc.*, vol. 41, part II, FJCC 1972.

[Wulf and Harbison, 77] Wulf, W. A. and S. Harbison, "Reflections in a pool of processors," Tech Report, Dept. Computer Science, Carnegie-Mellon University, Pittsburgh, PA., Nov. 1977.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CMU-CS-78-138	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) THE SWITCHING STRUCTURE AND ADDRESSING ARCHITECTURE OF AN EXTENSIBLE MULTIPROCESSOR: CM*		5. TYPE OF REPORT & PERIOD COVERED Interim 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Richard James Swan		8. CONTRACT OR GRANT NUMBER(s) N00014-77-C-0500
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Dept. Schenley Park, PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Arlington, VA 22217		12. REPORT DATE August 1978 13. NUMBER OF PAGES 235
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same as above		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Cm* is an extensible, multiprocessor computer system with a hierarchical structure. A 10 processor pilot system, constructed in the Computer Science Department, has been in operation for a year. A 50 processor system will be operational late in 1978. The hardware structure will support on the order of 10,000 processors. A major overall design objective is to efficiently support close cooperation between large numbers of concurrently executing processes.

→ next page

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601 |

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

An important component of multiprocessor systems is the switching structure which allows processors to access shared memory. The effective, maximum size (in number of processors) of multiprocessors described in the literature is limited. This limitation comes either through saturation of the access path to shared memory, or through the rapid growth of switch cost. The hierarchical switching structure of Cm* offers, in principle, indefinite extensibility of processing power, memory capacity and communication bandwidth. The cost of the interconnection structure grows approximately linearly with system size. Effective use of the structure depends on suitable decompositions of applications. We give the motivations for the switching structure and describe numerous techniques used to allow a high performance, cost-effective, deadlock free, switch realization.

The addressing architecture of a computer system has a strong influence on its programmability. The addressing architecture is important both at the level of naming operands, for example distinguishing between stack and register oriented designs, and in naming larger conceptual entities, such as segments. The former level is determined primarily by the instruction set design. The latter level is primarily determined by the operating system. The work in this thesis is concerned with understanding addressing architecture at the level of naming and sharing objects. An object is the smallest logical collection of information (for example a segment) which is independently manipulated by the operating system and can be directly accessed by a program.

We present a simple model which allows the systematic analysis and comparison of addressing architectures. The creation of an independent addressing environment when each subprogram is invoked (as in HYDRA and FAMOS) is compared to process-based addressing (as used in Multics and CAP). We show that independent addressing environments for each subprogram allows greater flexibility in the time of name binding, has fewer levels of indirection, and is better suited to multiprocessor systems.

Close cooperation and communication between processes requires the sharing of objects--shared access to data, semaphores, mailboxes, etc. Proper support of this sharing (particularly in a system with many, concurrently executing processes) is shown to have a substantial impact on the switching structure and addressing hardware of a multiprocessor.

As viewed from an individual processor, Cm* has a non-homogeneous structure. The access path and access time from a processor to a particular memory depends on their relative position in the structure. This inhomogenity could reduce the programmability of the system. However, the Cm* addressing architecture presents to the programmer a uniform, system wide, segmented address space. Segments are typed, protected objects. All references, by both user and kernel programs, are via capabilities. Capabilities may be passed between processes via mailbox segments. This allows the protected passing of messages, including large data structures represented by a capability list, without copying. Numerous facilities are provided in the architecture to support operating system primitives. Special provision is made to allow the movement of a segment without requiring every processor to update its mapping tables.